Analyse, Entwurf und Generierung von Rollen- und Variantenmodellen

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

DIPL.-INFORM. JÖRG BAUMGART

geboren in Darmstadt

Referenten:

PROF. DR. PETER KAMMERER

PROF. DR. MIRA MEZINI

Datum der Einreichung: 3. September 2003

Datum der mündlichen Prüfung: 16. Oktober 2003

Darmstädter Dissertationen D17

Vorwort

Bei der Anfertigung der vorliegenden Arbeit haben mir viele Personen ihre Unterstützung zukommen lassen, bei denen ich mich an dieser Stelle ganz herzlich bedanken möchte.

Mein besonderer Dank gilt den beiden Referenten dieser Arbeit, Herrn Prof. Dr. Peter Kammerer und Frau Prof. Dr. Mira Mezini, sowie Herrn Prof. Dr. Bernd Freisleben für die wertvollen Anregungen und Diskussionen. Weiterhin möchte ich mich bei Herrn Prof. Dr. Rolf Hoffmann für die Übernahme des Vorsitzes der Prüfungskommission bedanken.

Für die erfolgreiche Durchführung einer Arbeit spielt die Arbeitsatmosphäre eine entscheidende Rolle: die Fachgebiete Betriebssysteme, Rechnerarchitektur, Softwaretechnik und Parallele Systeme haben hier ein nahezu perfektes Umfeld geboten. An erster Stelle ist Gudrun Jörs zu nennen, deren unnachahmliches Organisationstalent auch unmöglich erscheinende Dinge möglich macht – Gudrun, vielen Dank für alles. Viele meiner Kollegen sind zu Freunden geworden und haben wesentlich zum Erfolg dieser Arbeit beigetragen: Thomas Barth, Michael Eichberg, Wolfgang Heenes, Henning Pagnia, Oliver Theel, Renato Vinga-Martins und Said Zahedani – vielen Dank für Eure Hilfe.

Besondere „Highlights“ waren immer die Kaffeerunden, Feiern, Betriebsausflüge und Workshops mit den Mitgliedern und Freunden der Fachgebiete Udo Arnold, Klaus Balser, Christoph Bockisch, Christoph Bußler, Vasian Cepa, Torsten Felzer, Ingo Fenske, Thomas Friemel, Felix Gärtner, Mathias Halbach, Philipp Harrschar, Peter Hartmann, Michael Haupt, Andreas Heck, Christian Hochberger, Ralph Jansen, Marion Kielmann, Thilo Kielmann, Sven Kloppenburg, Thomas Kunkelmann, Christoph Liebig, Marie-Luise Moschgath, Klaus Ostermann, Shadi Rifai, Guido Rößling, Ralf Schneider, Hartmut Vogler, Klaus-Peter Völkmann und Heidrun Werner.

Für ihre Unterstützung seitens der Berufsakademie Mannheim möchte ich mich bei Herrn Prof. Windel und Herrn Prof. Dr. Welter bedanken.

Allen Freunden und Bekannten, die mir während der verschiedenen Phasen der Arbeit zur Seite standen, bin ich ebenfalls zu Dank verpflichtet.

Mein ganz besonderer Dank gilt meiner Familie, die mir immer den notwendigen Rückhalt und jede nur erdenkliche Unterstützung zuteil werden ließ.

Darmstadt, im Oktober 2003 Jörg Baumgart

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADL	Aspect Description Language
AOP	Aspect-Oriented Programming
API	Application Programming Interface
BFL	Base Functionality Language
bzw.	beziehungsweise
DSC	Description
d.h.	das heißt
DOOR	Dynamic Object-Oriented database programming language with Roles
EDV	Elektronische Datenverarbeitung
EJB	Enterprise JavaBean
ERP	Enterprise Resource Planning
GUI	Graphical User Interface
ITHACA	An Integrated Toolkit for Highly Advanced Computer Applications
J2EE	JAVA 2 Enterprise Edition
MVC	Model View Controller
OID	Object Identifier
OL	Output Language
OMT	Object Modeling Technique
OOA	Objektorientierte Analyse
OOE	Objektorientierter Entwurf
OORAM	Object-Oriented Role Analysis and Modeling
OORASS	Object-Oriented Role Analysis, Synthesis and Structuring
OOSE	Object-Oriented Software Engineering
ORM	Object Role Model
ORM	Objects with Roles Model
PIN	Personal Identification Number
ROLEEP	Role Based Evolutionary Programming

UML	Unified Modeling Language
vgl.	vergleiche
XML	Extensible Markup Language
XPATH	XML PATH
XSL	The Extensible Stylesheet Language Family
XSL-FO	XSL FORMATTING OBJECTS
XSLT	XSL TRANSFORMATION
z.B.	zum Beispiel

Abbildungsverzeichnis

Programmverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

„Die Software-Systeme von heute sind die Altlasten von morgen“ – diese Aussage (vgl. [Bal96, S. 34]), welche eigentlich jeden EDV[1]-Verantwortlichen zutiefst beunruhigen müßte, sagt aber nicht aus, daß bei der Konzeption eines Software-Systems schon dessen Entsorgbarkeit eine zentrale Anforderung darstellt[2]. Stattdessen muß durch geeignete Software-Entwicklungsprozesse und Software-Strukturen sichergestellt werden, daß ein Software-System an neue Anforderungen angepaßt werden kann. Ein weiteres Problem stellen die hohen Entwicklungskosten der Software-Erstellung dar, wobei ein großer Anteil auf die Implementierung und die Wartung der Software entfällt.

Um die Qualität des Software-Entwicklungsprozesses zu erhöhen, ist es wichtig, ein Prozeßmodell zu verwenden, das eine möglichst weitgehende Integration der Phasen Analyse, Entwurf und Implementierung erlaubt. Daher spielt heute das objektorientierte Modell eine zentrale Rolle, da die grundlegenden Konzepte in den drei Phasen durchgängig unterstützt werden. In jeder Phase wird ein objektorientiertes Modell des Software-Systems beschrieben. Auf der Analyseebene ist ausschließlich die für den Anwender sichtbare Funktionalität entscheidend. Den nächsten Schritt stellt die Transformation des Analysemodells in ein Entwurfsmodell dar, wobei jetzt die technischen Randbedingungen zu berücksichtigen sind. Die letzte Transformation ist die Implementierung. Betrachtet man die Beschreibungstechniken des Software-Systems innerhalb der einzelnen Phasen, so fällt auf, daß wesentliche Beschreibungselemente der Analyse- und Entwurfsphase in der Implementierung nicht mehr explizit zu erkennen sind. Zwar sind auf allen Ebenen die Konzepte Klasse, Attribut, Operation und Vererbung vorhanden, so daß sich die statische Struktur des Analysemodells relativ gut in der Implementierung wiederfinden läßt, die in der Analyse beschriebene dynamische Struktur eines Systems besitzt jedoch in der Implementierung kein direktes Gegenstück mehr. Weder Anwendungsfalldiagramme noch Sequenzdiagramme oder Zustandsautomaten sind in der Implementierungsebene als eigenständige Einheiten vorhanden. Damit entsteht das Problem, die Konsistenz der Ebenen sicherzustellen. Dies betrifft sowohl die erstmalige Erstellung eines Software-Systems als auch dessen Weiterentwicklung. Geht man davon aus, daß in der Analyseebene das Systemverhalten korrekt beschrieben wurde, dann bedeutet eine Inkonsistenz, daß das Software-System seine Spezifikation nicht erfüllt. Um die Konsistenz der Beschreibungsebenen zu gewährleisten,

[1] EDV: Elektronische Datenverarbeitung

[2] Allerdings sollte auch dieser Aspekt berücksichtigt werden, da das vollständige Entfernen eines Programms von einem System nicht unproblematisch ist.

werden zunehmend Generierungstechniken eingesetzt. Diese spielen z.B. im User-Interface-Bereich eine wichtige Rolle. Durch die Entwicklung konsistenter Gesamtmodelle und den Einsatz von Generierungstechniken läßt sich der Entwicklungs- und Wartungsaufwand reduzieren, was zu einer deutlichen Kosteneinsparung führt.

Ziel dieser Arbeit ist es, für zwei wichtige Modellierungsbereiche, die Rollen- und die Variantenmodellierung, eine möglichst weitgehende Integration der drei Software-Entwicklungsphasen zu erreichen.

Rollenmodelle treten sehr häufig bei der Spezifikation langlebiger Objekte auf. Ein typisches Beispiel sind die Rollen, die eine Person im Laufe ihres Arbeitslebens annehmen und auch wieder aufgeben kann oder muß. Hier reichen die traditionellen, objektorientierten Konzepte wie Klassenbildung und Vererbung nicht aus, um den gewünschten Ausschnitt der realen Welt präzise in ein objektorientiertes Systemmodell abzubilden. Die Schwierigkeit besteht darin, daß durch die Übernahme einer Rolle ein Objekt zur Laufzeit eine zusätzliche Funktionalität erhält. Diese Funktionalität spiegelt sich darin wider, daß ein Objekt für den Zeitraum des Rollenbesitzes an seiner Schnittstelle zusätzliche Operationen zur Verfügung stellt. Dies entspricht einer dynamischen Erweiterung des Objekttyps. Zum Thema Rollenmodelle gibt es in der Literatur sehr viele Ansätze, die sich zunächst mit der konzeptuellen Struktur eines Rollenmodells beschäftigen. Nahezu alle Modelle sind auf das Konzept der Einfachvererbung eingeschränkt, d.h. eine Rolle besitzt (maximal) eine Oberrolle. Da es aber sinnvolle Anwendungsbeispiele gibt, die auf der Analyseebene eine Mehrfachvererbung von Rollen verwenden, sollte ein Rollenkonzept die Eigenschaft der Mehrfachvererbung nicht von vornherein ausschließen. Daher wird in dieser Arbeit auf der Analyseebene ein Rollenmodell mit Mehrfachvererbung vorausgesetzt. Um eine konsistente Transformation des Rollenmodells in die Implementierungsebene zu erreichen, wird ein Generierungsprozeß für Rollenmodelle realisiert.

Einen weiteren wichtigen Modellierungsbereich stellen Variantenmodelle dar. Hier geht es um Objekte, die aus einem Grundobjekt bestehen, in das andere Objekte ein- und ausgebaut werden können. Ein typisches Beispiel sind die Varianten eines Autos. Das Auto stellt eine Grundfunktionalität zur Verfügung, die durch den Einbau weiterer Komponenten (z.B. ein Schiebedach, ein Radio oder ein Navigationssystem) erweitert werden kann. Der Ein- und Ausbau von Komponenten kann dabei Einfluß auf die Struktur bestimmter Operationen nehmen. So wird beim Abstellen des Motors auch automatisch das Radio ausgeschaltet. Ein anderes Beispiel ist das Schließen des Schiebedachs und der Fenster, wenn die Fahrertür abgeschlossen wird. Da die hier beschriebene Variantenbildung auf einer Objektkomposition im Sinne einer *Whole-Part*-Beziehung aufbaut, sind Rollenmodelle für deren Beschreibung und Umsetzung nicht geeignet, weil diese eine *ist-ein*-Beziehung als Grundlage besitzen. Auf der Analyseebene wird die statische Struktur eines Variantenmodells über Kompositionsbeziehungen beschrieben, während die Operationen in Abhänigkeit der aktuellen Objektstruktur sehr gut mit Sequenzdiagrammen modelliert werden können. Um die Variantenmodelle in ein konsistentes Implementierungsmodell abzubilden, wird auch hier wieder ein Generierungsprozeß verwendet. Für die Spezifikation der Struktur wird eine einfache Beschreibungssprache definiert, die den statischen Aspekt des Modells festlegt. Daher müssen bei der Umsetzung des Variantenmodells bestimmte dynamische Aspekte manuell implementiert werden. Um hier eine möglichst direkte Umsetzung des dynamischen Modells der Analyseebene zu erreichen, wird für ein Variantenmodell eine Grundstruktur generiert, die die Aspekte *Strukturbeschreibung*, *Objektfunktionalität*, *globaler Kontrollfluß* und *Konfigurationswissen* auch auf der Implementie-

rungsebene strikt voneinander trennt. Damit finden sich die zentralen Elemente der Modellbeschreibung aus der Analyseebene (Klasse, Kompositionsbeziehung und Sequenzdiagramm) direkt auf der Implementierungsebene wieder, was die Sicherstellung der Konsistenz zwischen der Analyse-, der Entwurfs- und der Implementierungsebene deutlich vereinfacht.

Die weiteren Kapitel dieser Arbeit besitzen die folgende Struktur:

Kapitel 2 beschreibt den Software-Entwicklungsprozeß. Neben einer kurzen Vorstellung der einzelnen Phasen der Software-Entwicklung wird auf Prozeßmodelle und Software-Architekturen eingegangen. Der dort beschriebene Balancierte Makroprozeß wird für die Analyse der Funktionalität der neuen Rollen- und Variantenmodelle verwendet. Bei den Variantenmodellen wird außerdem davon ausgegangen, daß der Anwendungsentwickler für ein Variantenmodell diesen Prozeß verwendet. Daher ist in diesem Fall der Balancierte Makroprozeß der Ausgangspunkt für die Frage, welche Strukturen der Analyseebene sich auf der Implementierungsebene wiederfinden lassen sollten.

Kapitel 3 ermittelt an zwei Beispielen die Anforderungen an das neue Rollenkonzept. Auf der Ebene der Systemanalyse sollte das Konzept die Formulierung von Rollenmodellen erlauben, die sich strukturell über azyklische Graphen beschreiben lassen. Daneben müssen Restriktionen spezifizierbar sein und es sollte eine vollständige Integration in den Software-Entwicklungsprozeß unterstützt werden.

In Kapitel 4 werden alternative Ansätze zur Definition und Realisierung von Rollenmodellen vorgestellt und bewertet. Die betrachteten Rollenmodelle werden dabei in zwei Klassen eingeteilt: Ansätze zur Modellierung des dynamischen Vererbungscharakters und Ansätze zur Modellierung von Objektkollaborationen. Dabei werden Rollenmodelle aus dem Datenbankumfeld bis hin zu Agentensystemen betrachtet.

Kapitel 5 stellt das neue Konzept zur Realisierung von Rollenmodellen mit Mehrfachvererbung vor. In einem iterativen Prozeß werden die Anforderungen ausgehend von einem Rollenmodell mit Einfachvererbung schrittweise auf der Analyseebene ergänzt und dann jeweils in einen Entwurf umgesetzt. Neben der Mehrfachvererbung werden auch Restriktionen und Mehrfachrollen berücksichtigt, d.h. eine Rolle kann auch mehrmals übernommen werden. Für das entstandene Gesamtmodell wird dann die entwickelte Software-Architektur des Rollenmodellgenerators vorgestellt. Außerdem wird die Integration des Rollenmodells in den Software-Entwicklungsprozeß aus Anwendersicht beschrieben.

Die Spezifikation der Anforderungen an ein Variantenmodell ist die Zielsetzung von Kapitel 6. Aus einem Beispiel werden sechs Anforderungen abgeleitet, die ein Variantenmodell erfüllen sollte.

In Kapitel 7 werden alternative Ansätze zur Unterstützung der Variantenbildung betrachtet und bewertet. Neben dem Vererbungskonzept und einigen darauf basierenden Weiterentwicklungen werden Ansätze vorgestellt, die auf einer Objektkomposition aufbauen. Hier erstreckt sich das Spektrum von der reinen Kompositionsbeziehung bis hin zu Komponentenmodellen.

Die Beschreibung und Entwicklung des neuen Konzepts zur Umsetzung von Variantenmodellen ist der Inhalt von Kapitel 8. Die Analyse wird in zwei Schritten ausgeführt. Zunächst wird die statische Struktur eines Variantenmodells mit zwei Ebenen betrachtet: In ein sogenanntes Chassis-Objekt können Komponentenobjekte eingebaut werden, die selbst keine Unterstruktur mehr im Sinne des Variantenmodells besitzen. Im zweiten Schritt werden Variantenmodelle mit

beliebig vielen Ebenen betrachtet. Die dynamische Struktur des Analysemodells sowie die statische und dynamische Struktur des Entwurfsmodells werden ebenfalls in jeweils zwei Schritten erarbeitet. Den Abschluß des Kapitels bildet die Beschreibung der Software-Architektur zur Generierung der Variantenmodelle und die Integration des Variantenmodells in den Software-Entwicklungsprozeß aus Anwendersicht.

Im neunten Kapitel wird eine Fallstudie zur Realisierung der Scheduling-Komponente eines dynamisch adaptierbaren Betriebssystems durch ein Variantenmodell vorgestellt. An dem Beispiel der Scheduling-Komponente wird zunächst das Konzept zur Erzeugung minimaler applikationsspezifischer Betriebssystemstrukturen durch eine dynamische Adaption erläutert. Danach wird gezeigt, wie dieses Konzept für die Scheduling-Komponente durch ein Variantenmodell beschrieben und umgesetzt werden kann.

Das letzte Kapitel enthält die Zusammenfassung und einen Ausblick.

Kapitel 2

Der Software-Entwicklungsprozeß

2.1 Einleitung

Der Lebenszyklus eines Software-Systems ist in Abb. 1 dargestellt.

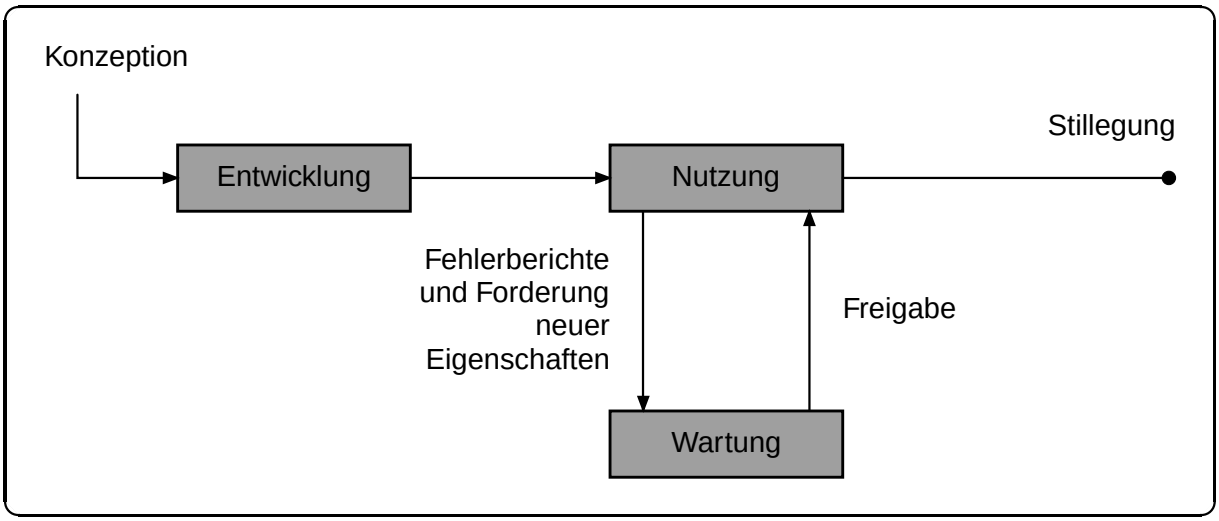


Abb. 1 Der Software-Lebenszyklus (vgl. [LL98, S. 408])

Durch die hohen Kosten für die Entwicklung von Individual-Software gibt es einen deutlichen Trend hin zur Nutzung von Standard-Software ([Cus89]). Diese zeichnet sich durch lange Entwicklungszeiten aber auch sehr lange Nutzungszeiten aus. Eine unmittelbare Folge ist, daß die Wartungsphase für derartige Software-Produkte immer wichtiger wird. Die Software muß der sich ständig ändernden Einsatzumgebung angepaßt werden. Diese Anpassungsprozesse verursachen oft 2/3 der Gesamtkosten des Software-Systems. Außerdem wird durch die langen Nutzungszeiten das Problem der Software-Altlasten immer größer, *da die Software-Produkte von heute die Altlasten von morgen sind* ([Bal96, S. 34]).

Für die Entwicklungsphase entstehen bei großen, langlebigen Software-Systemen zwei wesentliche Anforderungen:

- Eine arbeitsteilige Entwicklung ist zu unterstützen.

- Die entstandene Software muß so gut strukturiert sein, daß die notwendigen Funktionalitätsanpassungen und Fehlerkorrekturen effizient ausführbar sind.

Bei der Erfüllung dieser Anforderungen spielen die ***Phasen der Software-Entwicklung***, die ***Prozeßmodelle*** und die verwendete ***Software-Architektur*** eine wichtige Rolle. Auf diese drei Themengebiete wird in den folgenden Abschnitten eingegangen.

2.2 Die Phasen der Software-Entwicklung

Unabhängig von dem später gewählten Prozeßmodell läßt sich eine systematische Vorgehensweise zur Software-Entwicklung in vier große Phasen einteilen:

- Analyse,
- Entwurf,
- Implementierung und
- Test.

Die Analysephase, die auch als Definitionsphase bezeichnet wird, hat die Aufgabe, präzise festzulegen, welche Funktionalität das Software-System erfüllen muß. Dies beinhaltet auch eine Abgrenzung, welche Funktionalität *nicht* vorhanden sein soll. Die Analysephase selbst kann wieder in Teilphasen zerlegt werden. Typischerweise wird vom Auftraggeber ein Pflichtenheft vorgelegt, wobei als ein Muster für die verbale Anforderungsbeschreibung der *IEEE Guide to Software Requirements Specification* angesehen werden kann ([TM97]). Das Pflichtenheft sollte so abgefaßt sein, daß es als Basis eines juristischen Vertrags dienen kann. Es ist üblicherweise noch auf einem sehr hohen Abstraktionsniveau formuliert und stellt dann den Ausgangspunkt für eine detaillierte Analyse der geforderten Funktionalität dar. Die Analyse berücksichtigt nur das nach außen sichtbare Systemverhalten, da dieses die für den späteren Anwender nutzbare Funktionalität repräsentiert. Innerhalb der Analysephase sollen bewußt Entwurfs- und Implementierungsaspekte ausgeklammert werden, d.h. das Ziel der Analysephase ist eine *Beschreibung des WAS, nicht des WIE* ([Bal96, S. 105]). Ein weiteres wichtiges Merkmal der Analysephase besteht in der Annahme einer perfekten Technik ([MP88]). Die Spezifikation der Funktionalität erfolgt unabhängig von einer konkreten technischen Umgebung. Beispielsweise spielen Fragen bezüglich der zu verwendenden Hardware, einer Verteilung der Software, der ergonomischen Struktur der Benutzungsoberfläche oder der Form der Datenhaltung (z.B. Datei- oder Datenbanksystem) hier noch keine Rolle. Das Ergebnis der Analysephase wird oft als **Fachkonzeptmodell** bezeichnet. Parallel zum Fachkonzeptmodell sollte ein Prototyp der Benutzungsoberfläche entwickelt werden, der die Funktionalität des Fachkonzeptmodells widerspiegelt, aber natürlich weder eine interne Realisierung der Funktionalität noch eine Fähigkeit zur Datenspeicherung besitzt. Über diesen Prototyp wird dem Auftraggeber frühzeitig die Möglichkeit gegeben, sich ein Bild über das Fachkonzeptmodell zu machen ([ZBGK01, S. 91]).

Beim Übergang zu der Entwurfsphase wird die Annahme einer idealisierten Systemumgebung fallen gelassen. Der Entwurf hat die Aufgabe, das Fachkonzeptmodell unter den geforderten technischen Randbedingungen umzusetzen, ohne allerdings schon eine Implementierung vorzunehmen. Wichtige Ergebnisse des Entwurfs sind:

- Die vollständige Spezifikation der Funktionalität des Fachkonzeptmodells.
  Zum einen müssen jetzt auch alle *intern* notwendigen Funktionalitäten beschrieben werden, zum anderen ist zu klären, *wie* die Gesamtfunktionalität zu verwirklichen ist. Hier steht insbesondere die Algorithmenauswahl im Vordergrund.
- Die Gestaltung der Benutzungsoberfläche unter ergonomischen Gesichtspunkten.
- Die Auswahl eines Datenhaltungskonzepts.
  Hier wird entschieden, an welchen Stellen Dateien und Datenbanken einzusetzen sind und auch, welche Datenbankstruktur zu verwenden ist (objektorientiert, relational, hierarchisch, XML[1]-basiert).
- Die Verteilung der Software im Netz.
- Die Auswahl der Programmiersprachen und Klassenbibliotheken.

Das Ziel der Entwurfsphase ist es somit, präzise die Struktur der zu erstellenden Programme festzulegen.

Innerhalb der Implementierungsphase ist der Entwurf durch entsprechende Programme zu realisieren, wobei insbesondere vorgegebene Konventionen bezüglich der Namensgebung und der Dokumentation einzuhalten sind. Eine weitere wichtige Aufgabe der Implementierung ist die Integration der verschiedenen Programme zu einem Gesamtsystem.

Das Ziel der Testphase ist es sicherzustellen, daß die realisierte Systemfunktionalität die Anforderungen auch tatsächlich erfüllt. Einerseits müssen die einzelnen Programme ausführlich getestet werden, um nach Möglichkeit Fehler auf dieser Ebene auszuschließen. Andererseits sind aber oft auch bereits im Pflichtenheft Testfälle spezifiziert, welche die Gesamtfunktionalität des Anwendungssystems überprüfen sollen. Hiermit können Fehler im Zusammenspiel der verschiedenen Programme gefunden werden. Für die Einzelprogrammtests werden *White-Box*-Tests eingesetzt, bei denen die interne Struktur eines Programms vollständig überprüft werden sollte ([ZBGK01, S. 196], [Bal98, S. 400]). Bei der Kontrolle der Gesamtsystemfunktionalität finden *Black-Box*-Tests Anwendung, die innere technische Details ignorieren, und nur das nach außen sichtbare Verhalten des Systems oder eines Systemelements prüfen ([ZBGK01, S. 196], [Bal98, S. 426]).

2.3 Prozeßmodelle

Ein Prozeßmodell definiert für eine Software-Entwicklung einen festgelegten organisatorischen Rahmen ([Bal98, S. 98]).

Das erste Prozeßmodell, das *Stagewise Model*, wurde bereits 1956 von BENINGTON vorgeschlagen ([Ben56]) und bildet die Grundlage für das weit verbreitete Wasserfallmodell. Das *Stagewise Model* besitzt bezogen auf die in Abschnitt 2.2 (S. 6) vorgestellten Phasen die in Abb. 2 (S. 8) dargestellte Struktur. Die Phasen werden sequentiell durchlaufen, eine Rückkehr in eine bereits abgeschlossene Phase ist nicht vorgesehen. Da diese Vorgehensweise für größere Software-Projekte unrealistisch ist, wurde das Modell 1970 von ROYCE um die Möglichkeit erweitert, in eine direkt angrenzende Stufe zurückzukehren ([Roy70]). Dieses Modell wird in [Boe81] als **Wasserfallmodell** (*Waterfall Model*) bezeichnet und ist in Abb. 3 (S. 8) dargestellt.

[1] XML: Extensible Markup Language ([WWW00])

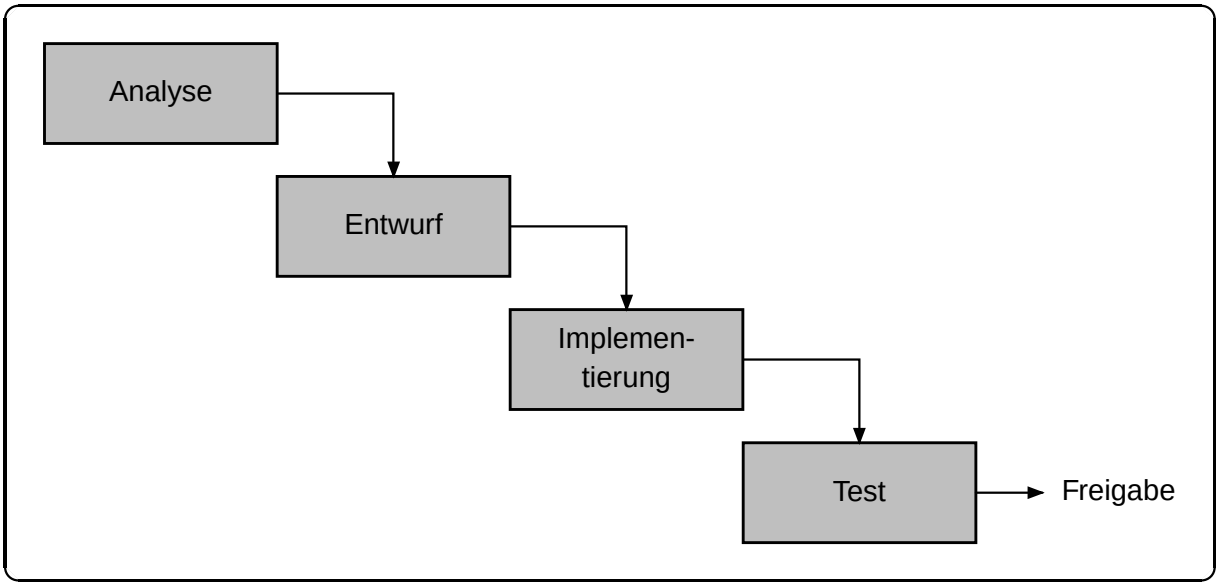


Abb. 2 Das Stagewise Model

Die Iterationen werden im Wasserfallmodell hauptsächlich zur Fehlerkorrektur verwendet. Um

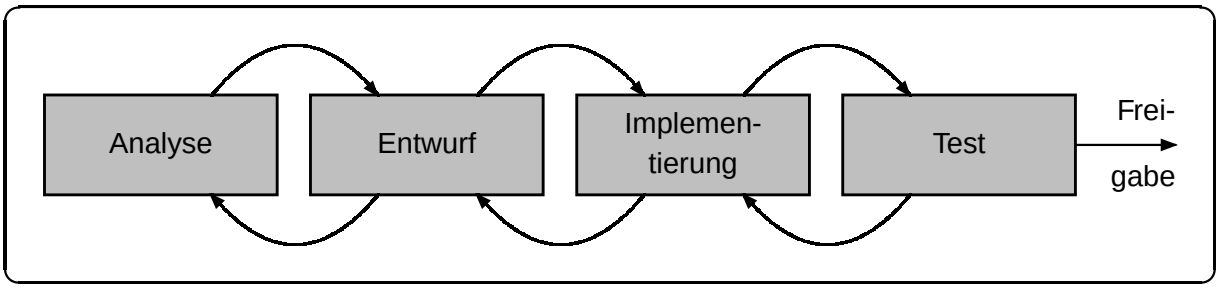


Abb. 3 Das Wasserfallmodell

das Konzept einer evolutionären Software-Entwicklung zu unterstützen, kann man das Wasserfallmodell in mehreren Gesamtiterationen durchlaufen. Abb. 4 (S. 9) zeigt das **Iterative Prozeßmodell** ([Som01, S. 52]). Im ersten Iterationsschritt werden die Basisanforderungen für ein Grundsystem ermittelt. Für diese Anforderungsmenge wird dann der gesamte Entwicklungsprozeß durchgeführt, bis ein lauffähiges Grundsystem vorliegt. In jedem neuen Iterationsschritt erfolgt die Identifikation weiterer Anforderungen, die dann in das jeweils existierende System zu integrieren sind. Der Prozeß terminiert, wenn alle Anforderungen umgesetzt wurden. Beispiele für iterative Prozeßmodelle sind das **Spiralmodell** ([Fri95, S. 48 f], [Som01, S. 53 ff]), das **Iterative Modell mit Prototyperstellung und Bewertung** ([LL98, S. 414 f]), der **Unified Process** ([ZBGK01], [Kru99]) und **Extreme Programming** ([Bec00], [Kir01]). Um den Anforderungen einer inkrementellen und evolutionären Systementwicklung zu genügen, eignet sich besonders gut ein objektorientiertes Prozeßmodell. Lassen sich Anforderungen inkrementell, d.h. additiv, in das bereits bestehende System integrieren, so erfolgt dies durch die Hinzunahme neuer Klassen. Müssen dagegen bei der Umsetzung neuer Anforderungen bestehende Funktionalitäten angepaßt werden, so bildet das Vererbungskonzept eine gute Ausgangsbasis. In [Bal99,

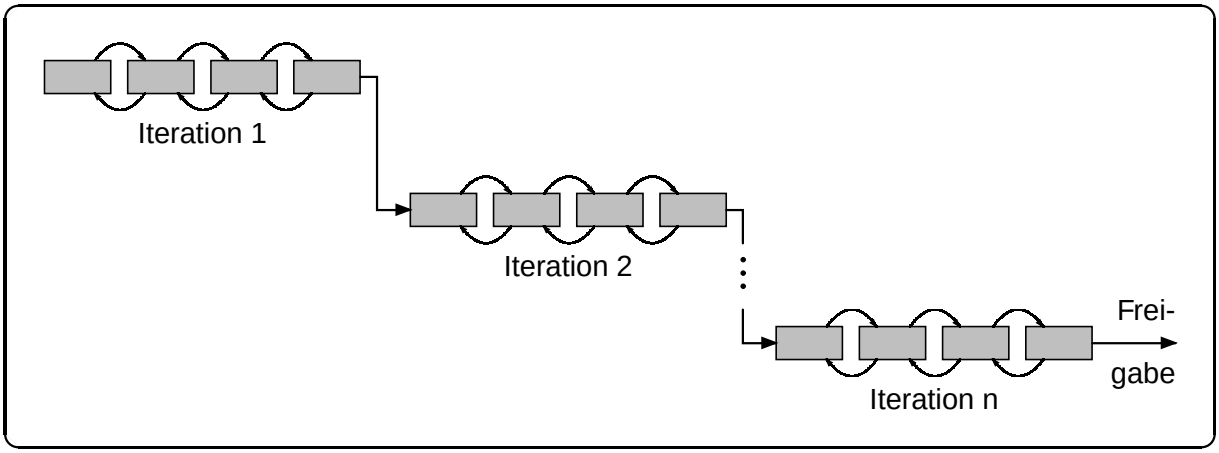


Abb. 4 Das Iterative Modell

S. 121 ff] wird als ein iteratives, objektorientiertes Prozeßmodell der **Balancierte Makroprozeß** vorgestellt (siehe Abb. 5).

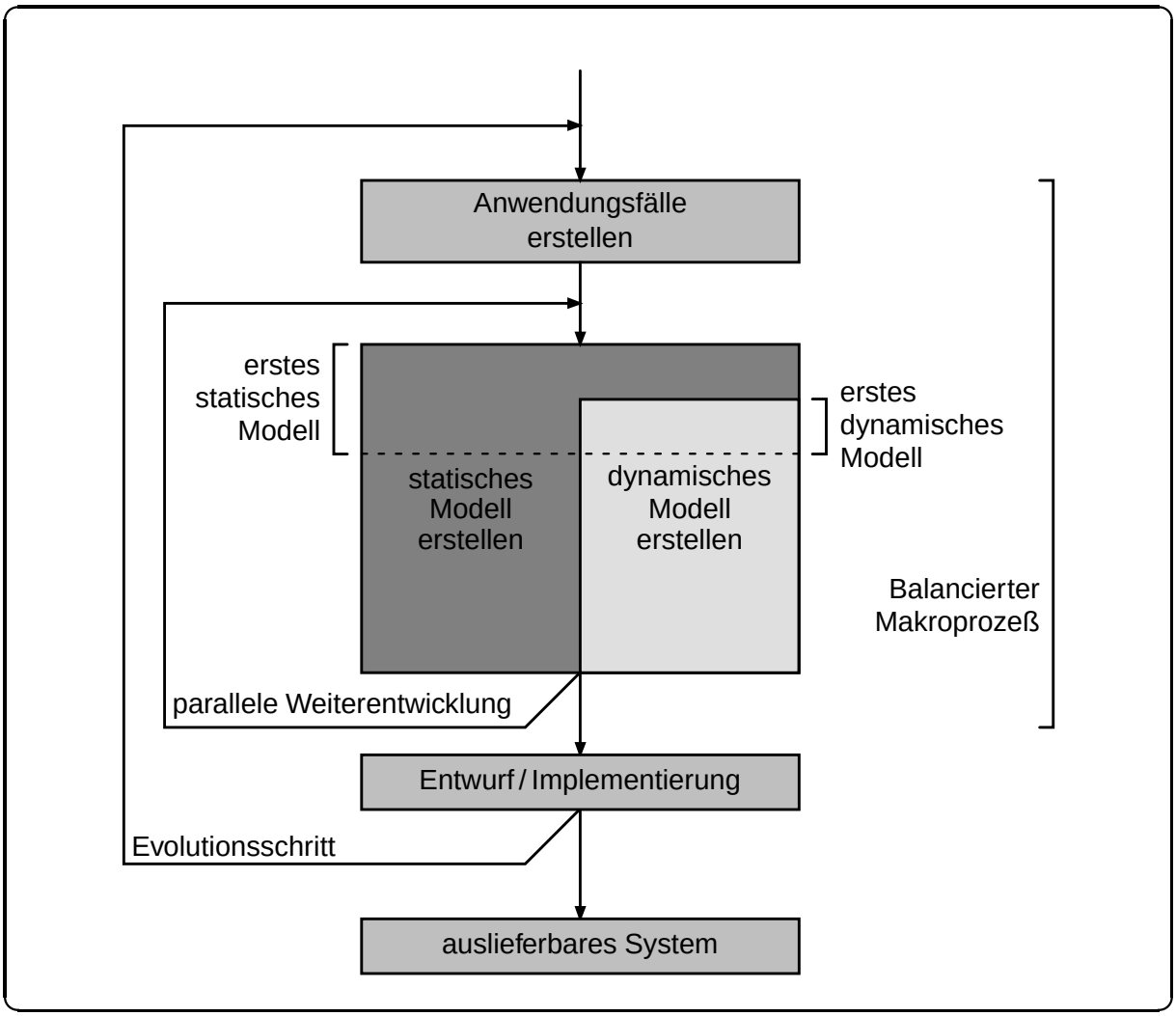


Abb. 5 Der Balancierte Makroprozeß

Ausgehend von den **Anwendungsfällen** (*Use Case*), die zur Spezifikation der Anforderungen an das Gesamtsystem dienen, erfolgt die Erstellung des **statischen** und des **dynamischen Modells**. Die Bestandteile der beiden Modelle sind in Abb. 6 dargestellt.

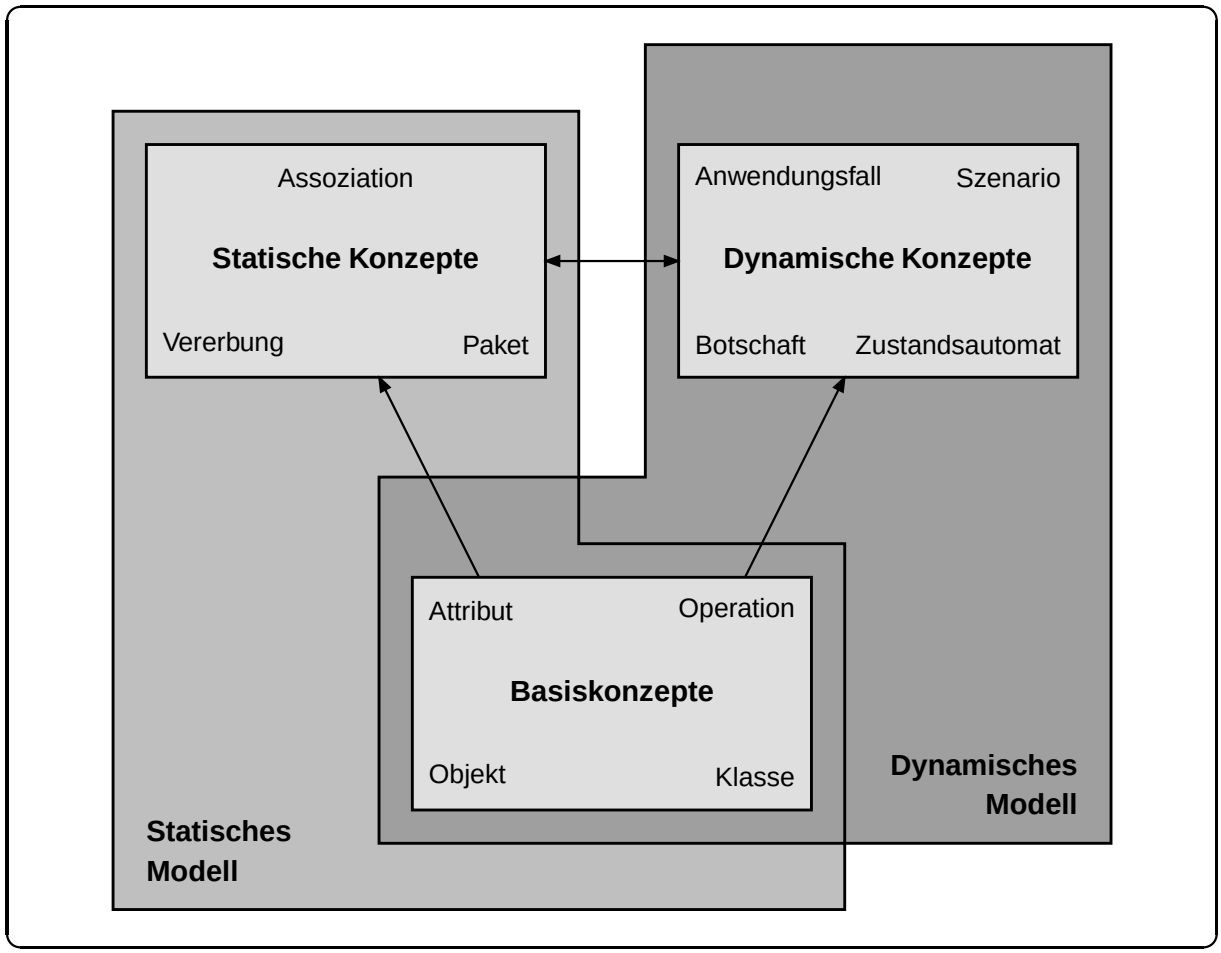


Abb. 6 Statisches und dynamisches Modell (vgl. [Bal99, S. 10])

2.4 Software-Architekturen

Die Auswahl einer Software-Architektur ist Aufgabe der Entwurfsphase. Für größere Systeme wird heute sehr häufig eine **Drei-Schichten-Architektur** (*Three-Tier-Architecture*) eingesetzt ([Bal99, S. 12 f], [Fow97a, S. 245 ff]). Die Gesamtsystemfunktionalität ist innerhalb der folgenden Schichten organisiert[2]:

- **Präsentationsschicht** (*Presentation Layer*): Realisierung der Benutzungsschnittstelle (*Graphical User Interface*, GUI).
- **Applikationsschicht** (*Application Layer*): Hier wird die eigentliche Programmfunktionalität umgesetzt. Diese Schicht wird in [Bal99] auch als **Fachkonzeptschicht** bezeichnet.
- **Datenzugriffsschicht** (*Data Access Layer*): In dieser Schicht werden die Zugriffsoperationen auf das verwendete Datenhaltungsmodell (z.B. Dateisystem oder Datenbanksystem) gekapselt.

Den Zusammenhang zwischen den Phasen Analyse, Entwurf und Implementierung sowie der Drei-Schichten-Architektur zeigt Abb. 7.

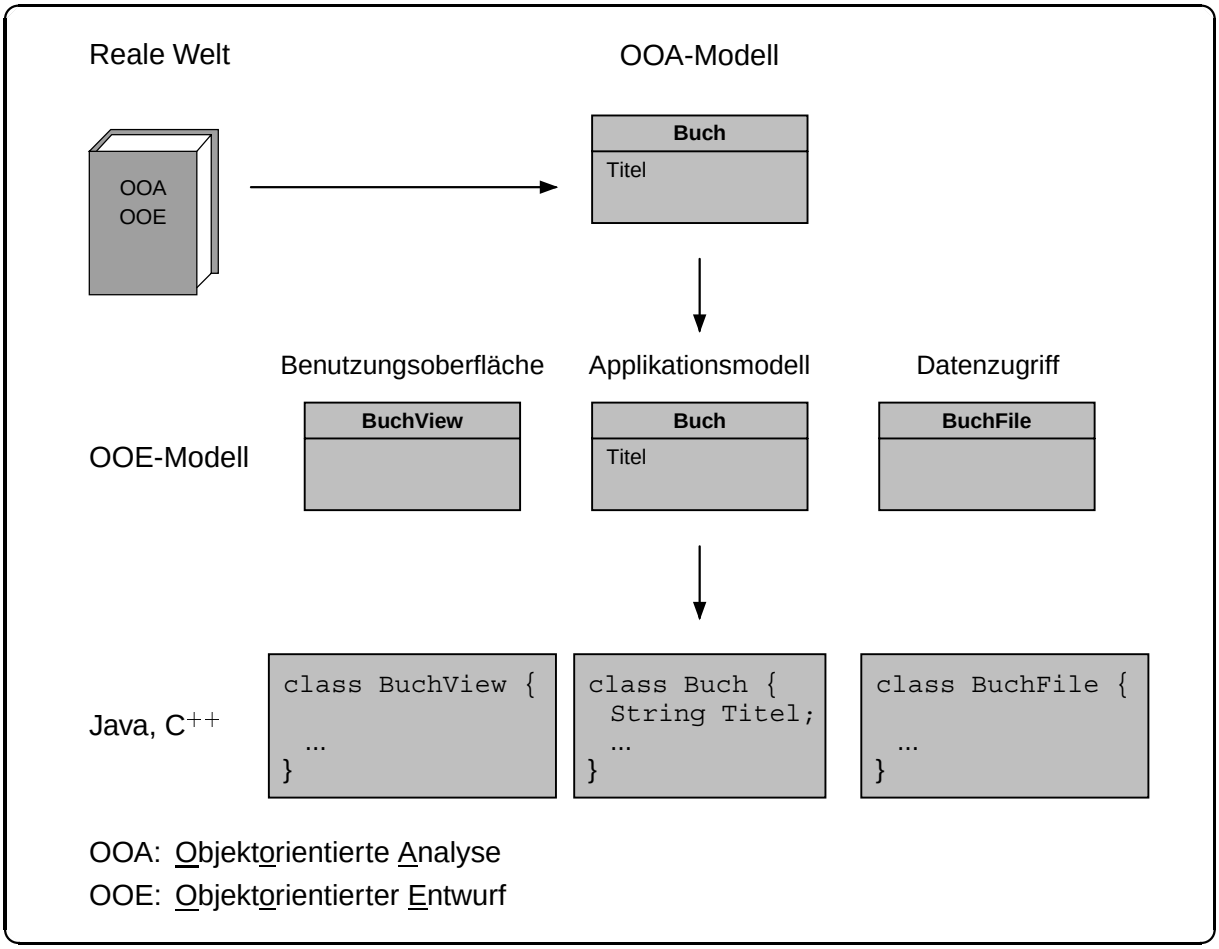


Abb. 7 Die Drei-Schichten-Architektur (vgl. [Bal99, S. 12])

[2] Eine Diskussion der Eigenschaften allgemeiner Schichtenarchitekturen findet sich in [Bal96, S. 639 ff].

Für die konkrete Verbindung zwischen den Klassen der Präsentations- und der Applikationsschicht wird in der Entwurfsphase oft das **Model-View-Controller**-Muster (MVC-Muster) verwendet ([KP88][3]). Abb. 8 beschreibt die Struktur des MVC-Musters.

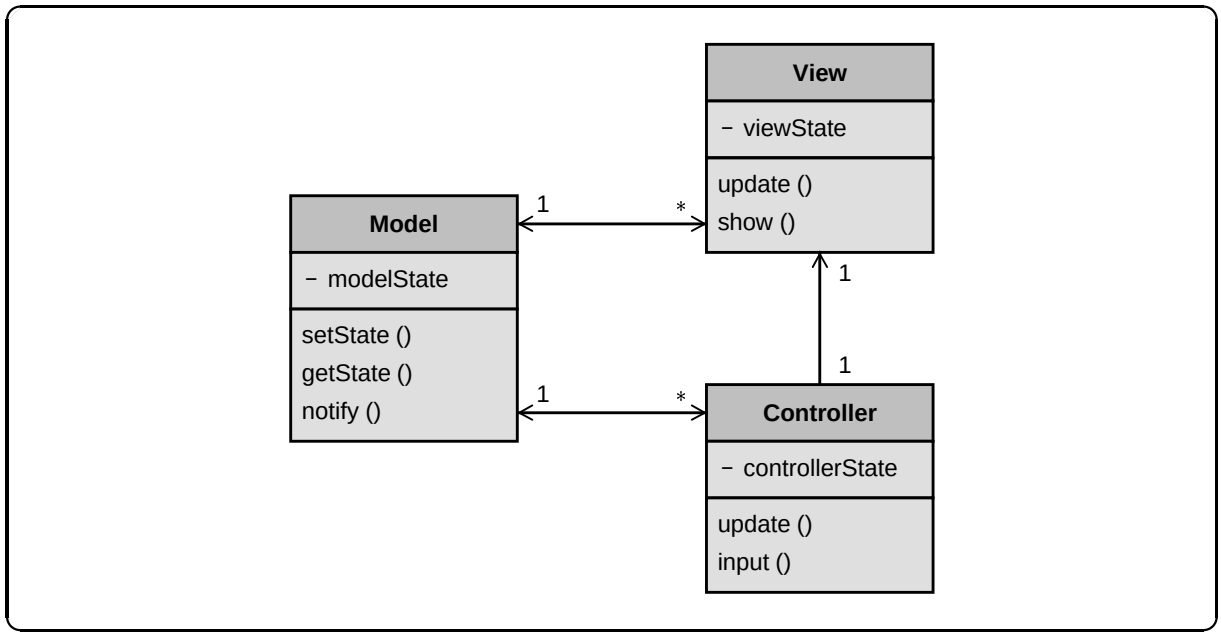


Abb. 8 | Das MVC-Muster (vgl. [Bal99, S. 373])

- Das **Model**-Objekt repräsentiert das Objekt der Applikationsschicht. Es darf nicht direkt auf die assoziierten View- und Controller-Objekte zugreifen. Es besitzt lediglich eine Liste aller Objekte, die von einer Zustandsänderung zu informieren sind.
- Ein **View**-Objekt gehört zur Präsentationsschicht. Die einem Model-Objekt zugeordneten View-Objekte übernehmen die Präsentation des Objektzustands. Die Verwendung mehrerer View-Objekte ist dann sinnvoll, wenn ein Objekt in verschiedenen Varianten präsentiert werden soll.
- Ein **Controller**-Objekt, das ebenfalls in der Präsentationsschicht liegt, bestimmt, wie auf Eingaben an der Benutzungsoberfläche zu reagieren ist. Jedem View-Objekt ist genau ein Controller-Objekt zugeordnet. Nach einer Eingabe wird diese gegebenenfalls (unter Verwendung der `show`-Operation) in den anderen View-Objekten dargestellt. Dann erfolgt über die `setState`-Operation die Weitergabe der Eingabe an das Model-Objekt. Falls die Eingabe zu einer Zustandsänderung führt, informiert das Model-Objekt alle assoziierten View- und Controller-Objekte, die daraufhin den neuen Zustand mit `getState` vom Model-Objekt abholen. Die View-Objekte stellen die Änderungen schließlich an der Benutzungsoberfläche dar.

[3] Die erste MVC-Version ist von TRYGVE REENSKAUG entwickelt worden, wie einem Zitat aus [RWL01, S. 318] zu entnehmen ist: *„The earliest example of an object-oriented framework is the **Model-View-Controller (MVC)** which I created when I was working with Adele Goldberg as a visiting scientist at the Xerox Palo Alto Research Center in 1978-79. It has been improved by Goldberg and her staff, and is now a powerful part of Objectworks\Smalltalk [GR83]“* .

Durch diese Architektur wird die Unabhängigkeit zwischen der Applikations- und der Präsentationsschicht realisiert. Müssen View- oder Controller-Objekte ausgetauscht werden, so ist die Funktionalität der Model-Objekte davon nicht betroffen.

2.5 Kapitelzusammenfassung

In diesem Kapitel wurden die Aufgaben der vier Software-Entwicklungsphasen Analyse, Entwurf, Implementierung und Test vorgestellt. Diese Phasen bilden die Grundlage für die Definition von Prozeßmodellen. Ein Prozeßmodell legt den organisatorischen Rahmen einer Software-Entwicklung fest. Es wurde kurz auf die historische Entwicklung von Prozeßmodellen eingegangen, die zu den heute verwendeten iterativen Prozeßmodellen geführt hat. Als Beispiel für ein objektorientiertes, iteratives Prozeßmodell wurde der Balancierte Makroprozeß vorgestellt, der als Prozeßmodell in dieser Arbeit verwendet wird. Neben dem Prozeßmodell spielt die Software-Architektur einer Anwendung eine entscheidende Rolle. Bei großen Software-Systemen bietet sich eine Drei-Schichten-Architektur an. Es wurde gezeigt, wie sich die Ergebnisse der Phasen Analyse, Entwurf und Implementierung auf eine Drei-Schichten-Architektur abbilden lassen. Hierbei stellt die Anwendung des Model-View-Controller-Musters ein wichtiges Konzept zur Entkopplung der Präsentations- und Anwendungsschicht dar.

Kapitel 3

Anforderungen an ein Rollenkonzept

3.1 Einleitung

Ein wesentliches Ziel der objektorientierten Modellierung besteht darin, einen Ausschnitt der realen Welt möglichst präzise in ein objektorientiertes Systemmodell abzubilden. Gerade bei langlebigen Objekten reichen die traditionellen, objektorientierten Konzepte wie Klassenbildung und Vererbung oft nicht aus, da sie eine Strukturveränderung eines Objekts zur Laufzeit nicht unterstützen. In diesem Kapitel wird an zwei Beispielen gezeigt, wie sich ein Rollenkonzept zur Modellierung langlebiger Objekte einsetzen läßt und welche Anforderungen an ein flexibles Rollenkonzept zu stellen sind.

3.2 Modellierung mit Rollen

Das klassische objektorientierte Modell beschreibt die gemeinsame Struktur einer Menge von gleichartigen Objekten im Rahmen einer Klassendefinition. Jedes Objekt stellt die Instanz ***genau einer*** Klasse dar. Ist diese Klasse Bestandteil einer Vererbungshierarchie, so besitzt das Objekt zusätzlich alle in den Oberklassen definierten Eigenschaften. Wird ein Software-System modelliert, das sehr ***langlebige Objekte*** enthält, so ist die Einschränkung auf eine unveränderliche Objektstruktur oft unrealistisch. Insbesondere bei der Analyse großer Informationssysteme, die typischerweise als Datenbankanwendungen realisiert werden, treten Objekte auf, die im Laufe ihres Lebenszyklus unterschiedliche Rollen annehmen. Für den Zeitraum der Rollenübernahme verfügt das Objekt über neue Eigenschaften.

Abb. 9 (S. 16) zeigt einen Ausschnitt aus einer Rollenhierarchie für eine Universitätsverwaltung. Da die UML-Notation[1] die Spezifikation von Rollen nicht unterstützt (vgl. [Obj03]), wird hier eine eigene Rollenbeziehung eingeführt: Die Grundlage bildet der Pfeil für die Vererbungsbeziehung, dessen Pfeilspitze aber schwarz ausgefüllt gezeichnet wird. Das **Gesamtobjekt** *Person* setzt sich aus einem **Grundobjekt** *RolePerson* und verschiedenen **Rollen** (*RoleStudent*, *RoleUniMitarbeiter*, *RoleWissenschaftlicherMitarbeiter* und *RoleVerwaltungsangestellter*) zusammen. Das Grundobjekt, in der Folge auch als **Wurzelrolle** bezeichnet, definiert die Lebensdauer des Gesamtobjekts. Mit der Erzeugung der Wurzelrolle entsteht auch

[1] UML: Unified Modeling Language

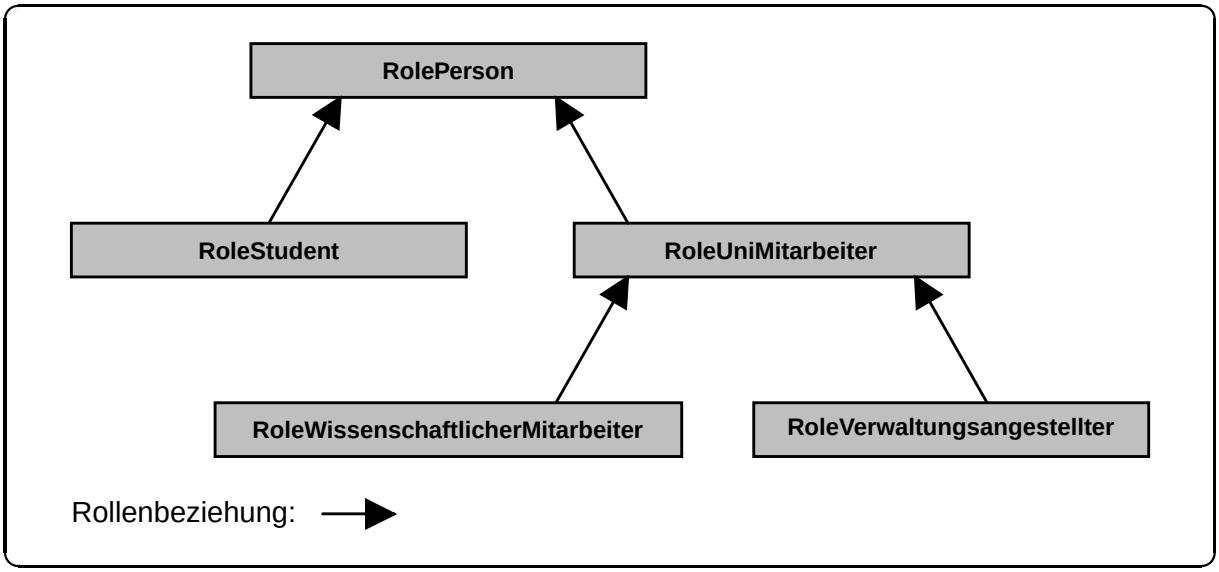


Abb. 9 Die Rollenhierarchie einer Universitätsverwaltung

das Gesamtobjekt und mit dem Löschen der Wurzelrolle hört das Gesamtobjekt auf zu existieren. Solange eine Person in der Universitätsverwaltung registriert ist, kann sie unterschiedliche Rollen annehmen. Sie kann zunächst studieren und später als Mitarbeiter an der Universität beschäftigt sein. Ihren Rollen entsprechend erwirbt sie neue Eigenschaften und verliert diese wieder, wenn sie die zugehörige Rolle abgibt. Die einzelnen Rollen dienen aber auch dazu, die Sicht auf das Gesamtobjekt einzuschränken. Falls eine Person gleichzeitig die Studenten- und die Mitarbeiterrolle besitzt, dann ist über die Studentenrolle die Matrikelnummer sichtbar, über die Mitarbeiterrolle aber nicht.

Weitere Eigenschaften finden sich in der Anforderungsliste an ein Rollenkonzept (vgl. [HK99, S. 58] sowie [KØ96a] und [GSR96]):

- AR1: Objekte können während ihrer Lebensphase mehrere Rollen annehmen und auch wieder ablegen.
- AR2: Ein Objekt kann eine Rolle mehrfach annehmen. Beispielsweise könnte ein Student die Rolle eines Tutors für mehrere Lehrveranstaltungen übernehmen.
- AR3: Objekte können Rollen *dynamisch* annehmen und wieder aufgeben.
- AR4: Jedes Objekt besitzt unabhängig von seinen aktuell vorhandenen Rollen eine eindeutige Objektidentifikation.
- AR5: Die einzelnen Rollen können *unabhängig* voneinander angenommen und wieder abgelegt werden.
- AR6: Mehrere Rollen können sich auf gemeinsame Daten sowie ein gemeinsames Verhalten beziehen.
- AR7: Ein bestimmtes Verhalten kann rollenspezifisch sein. Die Operation `telNr()` liefert für eine Person die Privatnummer, während sie für seine Rolle des Professors die Dienstnummer ermittelt.
- AR8: Rollen können die Sichtbarkeit von Eigenschaften beschränken. Das Gehalt eines Tutors ist in seiner Rolle als Student nicht sichtbar.

- AR9: Rollen können in einer eigenen Rollenhierarchie angeordnet werden.

In der Rollenhierarchie der Universitätsverwaltung kann jede Rollenbeziehung als ***ist-ein-Beziehung*** interpretiert werden: ein Student besitzt natürlich auch alle Eigenschaften einer Person, ebenso wie ein wissenschaftlicher Mitarbeiter ein Universitätsmitarbeiter und eine Person ist. Trotzdem ist das Vererbungskonzept, wie es z.B. in JAVA ([AGH00]) oder C^{++} ([Str00]) realisiert wurde, für eine Umsetzung eines Rollenkonzepts nicht geeignet, da beim Erzeugungszeitpunkt eines Objekts dessen endgültiger Typ festgelegt werden muß. Der Objekttyp ist danach während des Objektlebenszyklus nicht mehr änderbar. Um ein Rollenkonzept konsequent umzusetzen, ist ein Vererbungskonzept auf der Objektinstanzebene erforderlich, wie es z.B. in SELF angeboten wird [US87]. Dieses Konzept wurde allerdings in kommerziell genutzte objektorientierte Sprachen wie JAVA und C^{++}[2] nicht integriert.

Die Interpretation einer Rollenbeziehung als Vererbung auf Objektbasis stellt sich jedoch für die Systemmodellierung als zu restriktiv dar. Abb. 10 (S. 18) zeigt am Beispiel der Personalverwaltung einer Firma, daß auch andere Interpretationen sinnvoll sind. Ein vollzeitbeschäftigter, technischer Angestellter besitzt *ist-ein*-Beziehungen zu den Rollen Vollzeitkraft, Angestellter und Mitarbeiter. Er ***ist*** aber natürlich ***keine*** Technikerstelle, sondern ***übernimmt*** diese für einen bestimmten Zeitraum. Durch die Übernahme der Technikerstelle werden ihm folglich die Eigenschaften zugeordnet, die mit dieser Stelle verbunden sind. Daher ist es sinvoll, über die Rolle eines vollzeitbeschäftigten, technischen Angestellten auf die Eigenschaften der zugehörigen Planstelle zugreifen zu können. Alternativ wäre es natürlich auch möglich, zwischen den Rollen **Technikerstelle** und **TechnischerAngestellterVollzeit** eine Assoziationsbeziehung zu modellieren. Diese Variante ist insbesondere dann zu bevorzugen, wenn an der Methodenschnittstelle eines vollzeitbeschäftigten, technischen Angestellten nicht alle Methoden der Rolle Planstelle sichtbar sein sollen.

Das Beispiel der Personalverwaltung besitzt aber noch zwei weitere interessante Eigenschaften:

- Das Rollenmodell stellt keine Baumstruktur, sondern einen azyklischen Graphen mit zwei Wurzelrollen (Planstelle und Mitarbeiter) dar. Verläßt der technische Angestellte die Firma, dann wird zwar das Mitarbeiterobjekt mit seinen Instanzen der Rollen **Angestellter**, **Vollzeitkraft** und **TechnischerAngestellterVollzeit** gelöscht, die Instanz der Technikerplanstelle bleibt aber erhalten. Diese Rolleninstanz kann dann einem neuen Mitarbeiter zugeordnet werden.
- Im Gegensatz zu der Anforderung AR5 (vgl. S. 16) können bestimmte Rollen nicht unabhängig voneinander übernommen werden. Zwischen den Rollen **Vollzeitkraft** und **Teilzeitkraft** besteht eine ***exor***-Restriktion. Zu einem Zeitpunkt kann ein Angestellter nur eine der beiden Rollen übernehmen. Eine entsprechende Restriktion wurde für die Rollen **Kaufmannstelle** und **Technikerstelle** modelliert. Damit ist es möglich, einer Planstelle alternativ eine Technikerstelle oder eine Kaufmannstelle zuzuordnen, ohne daß die Zuordnung dauerhaft erfolgen muß. Hätte man an dieser Stelle die Klassenvererbung benutzt, dann müßte man sich beim Schaffen einer Planstelle entscheiden, ob sie dauerhaft für einen Techniker oder einen Kaufmann vorgesehen ist.

Um dem Systemanalytiker und auch dem Anwendungsentwickler ein flexibles Rollenmodell zur

[2] in Zukunft wohl auch C$^{\#}$ ([Arc01])

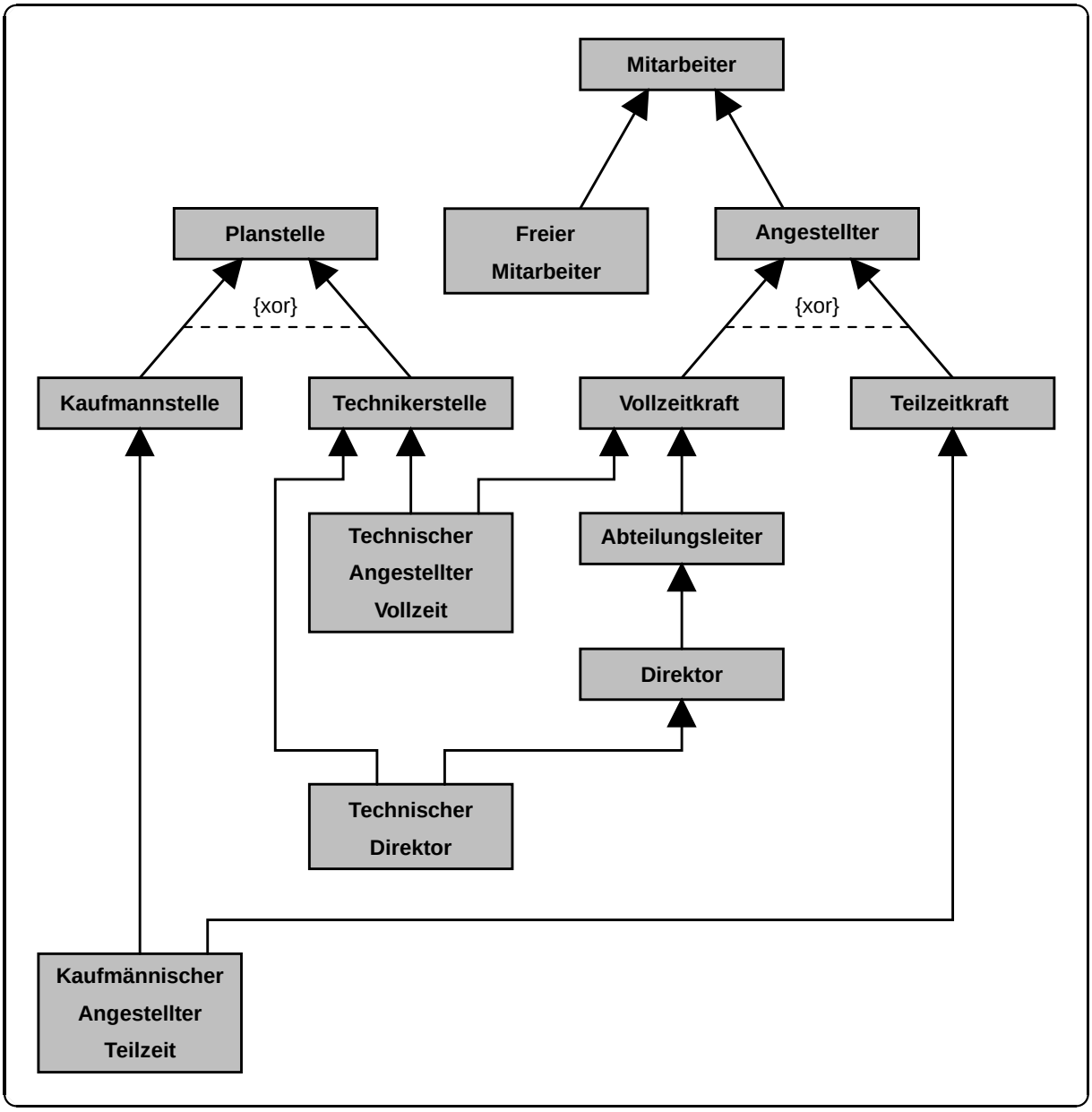


Abb. 10 Das Rollenmodell einer Personalverwaltung

Verfügung zu stellen, sind die folgenden Eigenschaften von zentraler Bedeutung:

- Über das Rollenmodell lassen sich Rollenbeziehungen beschreiben, die in ihrer Gesamtstruktur einem azyklischen Graphen entsprechen.
- Es müssen Restriktionen formulierbar sein, um eine möglichst präzise Abbildung der realen Welt im Rollenmodell vornehmen zu können.
- Das Rollenmodell ist vollständig in den Gesamtprozeß der Software-Entwicklung integriert. Um die Konsistenz zwischen dem Analyse-, Entwurfs- und Implementierungsmodell zu garantieren, sollte aus dem Analysemodell eine automatische Transformation in ein Entwurfs- und Implementierungsmodell stattfinden. Hierfür bietet sich ein Generierungsprozeß an.

3.3 Kapitelzusammenfassung

In diesem Kapitel wurde an zwei Beispielen gezeigt, daß ein Rollenkonzept sehr gut zur Modellierung von Objekten geeignet ist, die im Laufe ihrer Existenz Strukturänderungen unterliegen. Ein Ansatz zur Realisierung dynamischer Strukturänderungen ist die objektbasierte Vererbung. Diese kann als Grundlage für die Umsetzung eines Rollenkonzepts angesehen werden, sie ist aber allein nicht ausreichend, um allen Anforderungen an ein flexibles Rollenkonzept gerecht zu werden.

Auf der Ebene der Systemanalyse sollte ein Rollenkonzept die Erstellung von Rollenmodellen unterstützen, mit denen nicht nur Baumstrukturen, sondern auch azyklische Graphen darstellbar sind. Dies ist gleichbedeutend mit der Forderung, eine Mehrfachvererbung zwischen Rollen zuzulassen. Daneben ist es innerhalb der Analysephase wichtig, die Restriktionen, die in der realen Welt gelten, auch innerhalb des Rollenmodells spezifizieren zu können. Daher sollte ein Rollenkonzept die Modellierung von Restriktionen erlauben. Ein weiterer wichtiger Aspekt ist die vollständige Integration eines Rollenkonzepts in den Gesamtablauf der Systementwicklung. Hierbei ist die Konsistenz der Repräsentationen eines Rollenmodells auf der Analyse-, Entwurfs- und Implementierungsebene sicherzustellen. Diese Anforderung läßt sich durch die Anwendung eines Generierungsprozesses umsetzen: ausgehend von der abstrakten Beschreibung des Rollenmodells der Analyseebene sollte es möglich sein, die innerhalb der Implementierungsebene benötigten Strukturen automatisch zu erzeugen.

Kapitel 4

Existierende Ansätze zur Definition und Realisierung von Rollenkonzepten

4.1 Einleitung

Rollenmodelle können bezüglich ihres Anwendungsspektrums in zwei große Klassen eingeteilt werden. Die erste Klasse stellt den dynamischen Vererbungscharakter (dynamische Klassifikation) von Rollen in den Vordergrund. Hier geht es um die Frage, welche Strukturen sich über ein Rollenmodell definieren lassen. Ein wichtiges Beispiel stellen die Rollen dar, die Personen im Laufe ihres Lebens übernehmen können. Die zweite Klasse verwendet Rollen, um die Zusammenarbeit von Objekten zu beschreiben. Die Idee hierbei ist, daß Objekte bestimmte Rollen zeitweise übernehmen und damit die Möglichkeit erhalten, nach einem bestimmten, über die Rolle vorgegebenen Kommunikationsmuster mit den Rollen anderer Objekte zu interagieren. Ein typisches Beispiel ist die Modellierung von Verkäufer-Käufer-Beziehungen.

Für den folgenden Literaturüberblick wurde eine Einteilung der Rollenmodelle nach ihrem hauptsächlichen Einsatzbereich gewählt, wodurch mehr als zwei Klassen entstehen. Daraus ergibt sich, daß für einige Ansätze auch eine andere Zuordnung denkbar gewesen wäre. Beispielsweise wurde die OORAM-Methode (vgl. Abschnitt 4.4.1 (S. 42)) dem Thema Objektkollaborationen untergeordnet. Sie wird von ihren Autoren aber auch als Analysemethode bezeichnet.

Insgesamt kann die in dieser Arbeit verwendete Einteilung den beiden oben beschriebenen Klassen wie folgt zugeordnet werden:

- Modellierung des dynamischen Vererbungscharakters:
  - Rollenkonzepte aus dem Datenbankumfeld (Abschnitt 4.2 (S. 23)).
  - Das Rollenkonzept in der Analysephase (Abschnitt 4.3 (S. 26)).
  - Anwendung von Analyse- und Entwurfsmustern (Abschnitt 4.6 (S. 56)).
  - Realisierungskonzepte aus dem Programmiersprachenbereich (Abschnitt 4.7 (S. 69)).
- Modellierung von Objektkollaborationen:
  - Rollenkonzepte zur Modellierung von Objektkollaborationen (Abschnitt 4.4 (S. 42)).

- Rollenkonzepte im Umfeld von Agentensystemen (Abschnitt 4.5 (S. 51)).

An dieser Stelle soll auch noch kurz auf Arbeiten eingegangen werden, die nicht direkt unter dem Thema Rollenmodelle veröffentlicht wurden, aber bereits Konzepte enthalten, die in Rollenmodellen relevant sind und dort auch aufgegriffen wurden. SCIORE stellt in [Sci89] das Konzept der **Objektspezialisierung** vor, über das sich Vererbungshierarchien auf Objektbasis beschreiben lassen. Das Konzept wurde in das objektorientierte Datenbanksystem VISION integriert. Ebenfalls im Datenbankumfeld ist der **ASPECTS**-Ansatz von RICHARDSON und SCHWARZ entstanden ([RS91]). Objekte können mehrere Aspekte besitzen, die jedoch keinen Vererbungscharakter haben. Jeder Aspekt definiert eine eigene Objektsicht. Aspekte können dynamisch zur Laufzeit angenommen und wieder aufgegeben werden. Bei dem **CONTRACTS**-Modell von HELM, HOLLAND und GANGOPADHYAY ([HHG90]) liegt der Schwerpunkt auf der Formalisierung der Objektzusammenarbeit. Ein Contract identifiziert die Teilnehmer an einer Objektzusammenarbeit und spezifiziert, welche Variablen und externen Methodenschnittstellen die einzelnen Teilnehmer zur Verfügung stellen müssen. Außerdem wird festgelegt, in welcher Reihenfolge bestimmte Aktionen der Teilnehmer auszuführen sind und welche Invarianten während der Objektzusammenarbeit einzuhalten sind. HARRISON und OSSHER entwickeln in [HO93] das Konzept der **Subjektorientierten Programmierung** (*Subject-Oriented Programming*). Der dort definierte Subjektbegriff wird in den Arbeiten von KRISTENSEN und ØSTERBYE aufgegriffen (vgl. Abschnitt 4.3.2 (S. 27)).

4.2 Rollenkonzepte aus dem Datenbankumfeld

4.2.1 Einführung

Im Bereich der Wissensrepräsentation fand die Entwicklung von Rollenkonzepten schon sehr früh statt ([BW77], [GB80], [SB86]). Dagegen wurden im Datenbankumfeld Rollenkonzepte intensiv erst im Zusammenhang mit der Entwicklung objektorientierter Datenbanksysteme betrachtet, obwohl bereits 1977 BACHMANN und DAYA in [BD77] den ersten Vorschlag für die Integration eines Rollenkonzepts als Erweiterung des Netzwerkdatenbankmodells präsentiert haben. Rollen fanden dort für die Verhaltensmodellierung Verwendung, da es in dem klassischen Netzwerkdatenmodell wie auch im relationalen Datenbankmodell ([EN94]) noch keine Integration von Daten und Operationen gab.

Wichtige Ansätze für Rollenkonzepte aus dem objektorientierten Datenbankbereich werden in den folgenden Abschnitten vorgestellt.

4.2.2 Der Object-Role-Model-Ansatz (ORM) von PAPAZOGLOU und KRÄMER

Innerhalb des ORM-Ansatzes ([PJ00], [PK97], [Pap91], [PJA94]) werden Rollen verwendet, um es einer Applikation zu erlauben, eigene Sichten für ein bestehendes Datenbankschema zu definieren. Für eine Klasse des Datenbankschemas (z.B. Mitarbeiter) können von der Applikation unter Verwendung entsprechender Kommandos der Schemadefinitionssprache neue Rollenklassen definiert werden. Von den Rollenklassen können allerdings keine eigenen Objekte erzeugt werden. Stattdessen werden bereits vorhandene Objekte des Datenbankschemas automatisch über zuvor definierte Regeln den neuen Rollen zugeordnet. Sowohl Klassen als auch Rollenklassen verwalten die Mengen der Objektidentifikationen (OID) derjenigen Objekte, die zu der entsprechenden Klasse gehören. Wird eine neue Rollenklasse `RA` als Erweiterung einer bestehenden Klasse `A` erzeugt, werden die Objekte aus `A` überprüft, und alle OIDs nach `RA` übernommen, deren Objekte den entsprechenden Regeln zur Konstruktion von `RA` genügen. Ein Beispiel ist die Klassifikation von Kunden in die neuen Rollenklassen `GuteKunden` und `SchlechteKunden` anhand der Summe der vorgenommenen Einkäufe. Dies wird auch als *Objektmigration in eine Unterklasse* bezeichnet. Neben der Klassifikation während der Erzeugung einer Rollenklasse ist es möglich, Rollen zu wechseln. Derartige Wechsel werden üblicherweise bei der Ausführung eines Triggers durchgeführt. Trigger stellen in Datenbanken ein zentrales Konzept dar, um an bestimmte Ereignisse eine Verarbeitungsfunktionalität zu koppeln. Kauft beispielsweise ein schlechter Kunde Waren ein, wird über einen Trigger, der dem Attribut `SummeDerEinkaeufe` zugeordnet ist, bei jeder Attributänderung kontrolliert, ob der Wert die Grenze überschritten hat, damit der Kunde von der Rolle `SchlechterKunde` in die Rolle `GuterKunde` wechselt. Der Ansatz ist eng mit der Definition von Sichten (*View*) in objektorientierten Datenbanksystemen verwandt ([KK95]). Ein wesentlicher Unterschied besteht in der Verwendung der Objektidentifikationen. Während im ORM-Ansatz einem Objekt weiterhin nur eine einzige Identifikation zugeordnet ist, unabhängig davon, wieviele Rollen es besitzt, können Views als virtuelle Klassen aufgefaßt werden. Ein Objekt, für das eine neue *Sicht* definiert wird, erhält eine neue Objektidentifikation.

Der ORM-Ansatz besitzt zwei wesentliche Nachteile:

- Durch die Beibehaltung der Objektidentifikationen ist es nicht möglich, daß ein Objekt mehrere Rollen desselben Typs annehmen kann.
- Das Rollenkonzept wird als eine dynamische Erweiterung der Schemadefinition aufgefaßt. Somit liegt der Schwerpunkt in der Erweiterung der Schemadefinitionssprache. Es gibt keine Integration des Rollenkonzepts in eine Programmiersprache, die auch für die Erstellung von Anwendungen genutzt werden kann.

4.2.3 Der FIBONACCI-Ansatz

FIBONACCI ([ADG95], [ABGO93], [AGO95]) ist eine streng typisierte, objektorientierte Datenbankprogrammiersprache mit statischer Typprüfung. FIBONACCI integriert Konzepte konventioneller objektorientierter Programmiersprachen mit denen aus Datenbanksprachen und hat auf der Datenbankseite Strukturen der GALILEO-Sprache ([ACO85]) übernommen. Eine zentrale Erweiterung gegenüber GALILEO stellt die Aufnahme eines Rollenkonzepts dar. Ein Objekt kann als eine Art Container für Rollen aufgefaßt werden und besitzt eine eindeutige Identität. Die Funktionalität beschränkt sich auf einen Vergleichsoperator sowie Methoden zur Erzeugung und Verwaltung von Rollen. Der Zugriff auf ein Objekt findet immer über Rollen statt, welche die inhaltliche Funktionalität des Objekts modellieren. Der Objekttyp ist der Supertyp aller seiner Rollentypen. Bei der Deklaration eines neuen Objekttyps wird nur dessen Name spezifiziert. Ein Rollentyp dagegen legt fest, auf welche Nachrichten die Rolle reagiert. Eine Implementierung des Rollentyps muß dann für diese Nachrichten entsprechende Methoden bereitstellen. Für einen Rollentyp kann es mehrere Implementierungen geben. Die Rollentypen eines Objekttyps können als azyklischer Graph organisiert werden, wodurch eine Mehrfachvererbung bei Rollentypen unterstützt wird. Rollen können zur Laufzeit dynamisch übernommen werden. Für die Interpretation einer Nachricht, d.h. die Methodensuche, werden zwei Strategien zur Verfügung gestellt:

- *Upward Lookup*: Es wird zunächst überprüft, ob die Rolle, die diese Nachricht erhalten hat, die Methode besitzt. Falls nicht, werden die Oberrollen durchsucht. Da FIBONACCI über eine statische Typprüfung verfügt, ist garantiert, daß eine der Oberrollen die gewünschte Methode besitzt.
- *Double Lookup*: Zunächst werden alle Unterrollen durchsucht. Es wird mit derjenigen Unterrolle begonnen, die zeitlich gesehen als letztes von dem Objekt übernommen wurde. Wird bei den Unterrollen die Methode nicht gefunden, findet die *Upward-Lookup*-Strategie Anwendung.

Welche der beiden Strategien benutzt werden soll, hängt von der verwendeten Syntax eines Methodenaufrufs ab.

Als Nachteile des FIBONACCI-Ansatzes sind zu nennen:

- Ein Objekt kann eine Rolle nur einmal übernehmen.
- Es sind keine Restriktionen für die Erzeugung von Rollen definierbar.
- Das Löschen einer Rolle wird nicht über eine Verwaltungsoperation unterstützt. Eine Rolle wird erst dann gelöscht, wenn das zugehörige Objekt gelöscht wird. Dieses Löschen erfolgt implizit, wenn die letzte Referenz auf das Objekt gelöscht wurde.

Eine Formalisierung dieses Rollenmodells wird in [Ghe02] präsentiert. Das Rollenkonzept aus FIBONACCI wurde in die Sprache GALILEO 97 ([AAG00]) übernommen. Hier ist dann auch das explizite Löschen von Rollen vorhanden.

Die Idee, für die Methodensuche eine Zeitrelation zur Definition einer totalen Ordnung zwischen den Rollen eines Objekts zu nutzen, wird auch in dem CHIMERA-Datenmodell ([BG95]) verwendet. Neben der Reihenfolge, in der die einzelnen Rollen erzeugt wurden, kann hier alternativ der Definitionszeitpunkt der Klasse, die die Rolle beschreibt, ausgewählt werden.

4.2.4 Der DOOR-Ansatz

In [WCL97] und [WCL96] werden das Datenmodell und die Semantik der Datenbankprogrammiersprache DOOR[1] vorgestellt. Ein Objekt kann mehrere Rollen übernehmen, und eine Rolle selbst kann auch wieder Unterrollen besitzen. Im Unterschied zum FIBONACCI-Ansatz kann ein Objekt aber auch unabhängig von einer Rolle existieren. Neben der Objektidentifikation liegen eigene Identifikationen für die einzelnen Rollen vor, die dann wieder die Grundlage für die Methodensuche bilden. Als Identifikation für eine Rolle wird allerdings kein systemweit eindeutiger Bezeichner verwendet, sondern der Name der Rollenklasse. Da DOOR keine Mehrfachvererbung für Rollen anbietet, bildet ein Objekt zusammen mit seinen Rollen immer eine Baumstruktur. Insgesamt sind auf der Baumstruktur sieben Verfahren zur Methodensuche anwendbar. Die Methodensuche startet bei allen Varianten bei der referenzierten Rolle. Daher ist eine der FIBONACCI *Double-Lookup*-Strategie vergleichbare Strategie nicht vorhanden. Bei zwei Varianten wird der gesamte Baum traversiert, d.h. die Methode wird bei allen Rollen gesucht. In DOOR ist es damit möglich, daß beispielsweise für eine Person mit den Rollen Student und Vereinsmitglied eine Methode `getAddress` ausgehend von Student auch innerhalb der Rolle Vereinsmitglied gesucht wird. Diese Vorgehensweise steht jedoch im Gegensatz zu der üblichen Sichtweise, daß eine Rolle unabhängig von Rollen aus anderen Zweigen der Baumstruktur ist. Für die Reihenfolge der Rollenerzeugung können Restriktionen definiert werden. Außerdem ist es möglich, daß von einer Rolle mehrere Instanzen existieren. Um eine Rolleninstanz auszuwählen, kann neben dem Namen der Rollenklasse ein boolescher Ausdruck angegeben werden, der Eigenschaften für bestimmte Attribute der gewünschten Rolle spezifiziert. Wenn beispielsweise eine Person Mitglied in mehreren Vereinen ist und damit mehrere Instanzen der Rolle Klubmitglied besitzt, kann die Rolle über den Namen des Klubs gewählt werden. Falls mehrere Rollen den booleschen Ausdruck erfüllen, liefert das System nur eine Rolle zurück, die zudem noch zufällig ausgewählt wird. DOOR unterstützt explizit die Migration von Rollen durch eine eigene `role_move`-Operation.

Die Nachteile des DOOR-Ansatzes sind:

- Die Mehrfachvererbung von Rollen wird nicht unterstützt.
- Die Auswahl einer Rolleninstanz beim Vorliegen mehrerer Rolleninstanzen kann zu einem nicht deterministischen Ergebnis führen.
- Zwei Varianten der Methodensuche widersprechen den Sichtbarkeitsregeln von Methoden beim Rollenkonzept.

[1] DOOR: Dynamic Object-Oriented database programming language with Roles

4.3 Das Rollenkonzept in der Analysephase

4.3.1 Das Objects-with-Roles-Modell (ORM)

PERNICI stellt in [Per90] ein Rollenkonzept vor, dessen Schwerpunkt in der Modellierung des Verhaltens von langlebigen Objekten liegt. Dieser Ansatz wurde im Rahmen des ITHACA[2]-Projekts entwickelt ([PTMA89], [ANM+90]), welches die Bereitstellung von Entwicklungswerkzeugen für objektorientierte Anwendungen zum Ziel hatte. Das Rollenmodell besitzt die folgende Struktur:

- Ein Objekt, die *Basisrolle*, kann zur Laufzeit verschiedene Rollen aus einer vordefinierten Menge annehmen.
- Eine Rollendefinition (inclusive der Basisrolle) hat fünf Bestandteile:
  - Der Rollenname (*Role Name*).
  - Eine Menge von Eigenschaften (*Properties*), die als abstrakte Definition der Daten der Rolle dient.
  - Eine Zustandsmenge (*States*). Das Verhalten des Objekts, das diese Rolle zur Zeit übernommen hat, wird durch eine Folge von Zustandsübergängen aus der Zustandsmenge beschrieben.
  - Eine Nachrichtenmenge (*Messages*). Hier werden sowohl die Nachrichten aufgeführt, auf die ein Rollenobjekt reagiert, als auch die Nachrichten, die ein Rollenobjekt versenden kann. Es werden aber nur solche eingehenden Nachrichten angegeben, die relevant für die Verhaltensspezifikation des Rollenobjekts sind, d.h. die möglicherweise zu einem Zustandswechsel führen.
  - Eine Regelmenge (*Rules*). Die Regelmenge legt fest, welche Nachrichten in welchem Zustand gesendet bzw. empfangen werden können. Zusätzlich definiert die Regel den Folgezustand, der nach der Ausführung der Operation, die der Nachricht zugeordnet ist, eingenommen wird.

Für das Erzeugen und Löschen von Rollenobjektinstanzen existieren in der Basisrolle entsprechende Nachrichten. Prinzipiell können zu einem Zeitpunkt auch mehrere Instanzen desselben Rollentyps existieren. Wann welche Rollenobjektinstanzen erzeugt und gelöscht werden dürfen, wird durch entsprechende Regeln innerhalb der Basisrollendefinition festgelegt. Das Modell unterstützt auch eine Koordination von Rollenobjektinstanzen. Die Menge der erlaubten Zustandsübergänge einer Rollenobjektinstanz `RA` kann von dem aktuellen Zustand anderer Rollenobjektinstanzen abhängen. Diese Restriktionen werden ebenfalls wieder über entsprechende Regeln beschrieben.

Als Grundlage für eine Umsetzung dieses Rollenmodells werden in [Per90] im Abschnitt CONCLUDING REMARKS AND FUTURE WORK die objektorientierte Sprache COOL ([KMSW89]) und ein Objektkonfigurationsmechanismus ([KVN+89]) genannt, die beide innerhalb des ITHACA-Projekts entwickelt wurden. Eine zentrale Anforderung liegt dabei in der Realisierung des Regelauswertungsmechanismus. Bevor die Operation, die einer Nachricht zugeordnet ist, ausgeführt werden darf, müssen zunächst alle relevanten Regeln bestimmt werden, indem getestet wird, ob der aktuelle Zustand und die Nachricht zu der Regel passen. Sind mehrere Regeln

[2] ITHACA: An Integrated Toolkit for Highly Advanced Computer Applications

anwendbar, werden diese konzeptuell parallel ausgewertet. Allerdings wird keine Aussage darüber gemacht, welches der neue Zustand ist, wenn sich die Folgezustände der ausgeführten Regeln unterscheiden.[3]

Als wesentliche Eigenschaften des Rollenmodells lassen sich festhalten:

- Es werden nur zwei Hierarchieebenen unterstützt (Basisrolle und Rollen). Eine Rolle selbst kann keine weiteren Unterrollen annehmen.
- Die Formulierung von Restriktionen für das Rollenmodell erfolgt über Regeln. Damit sind prinzipiell beliebige Restriktionen realisierbar, allerdings ist die Überprüfung der Restriktionen sehr aufwendig, da zur Laufzeit immer erst die relevante Regelmenge zu bestimmen ist. Zusätzlich ist bei der Spezifikation eines Rollenmodells sicherzustellen, daß über die Regeln keine inkonsistenten Restriktionen formuliert wurden.
- Der Zustand eines Rollenobjekts kann Auswirkungen auf Zustandsänderungen anderer Rollenobjekte haben. Damit geht natürlich die Unabhängigkeit der aktuell existierenden Rollenobjekte untereinander verloren.
- Eine Basisrolle kann auf der konzeptuellen Ebene dynamisch Rollen annehmen und wieder ablegen. Wie die Realisierung dieser Eigenschaft mit COOL erfolgt, wird nicht näher beschrieben. Da COOL eine streng typisierte Sprache ist ([PTMA89]), muß einer der Ansätze verwendet werden, die in Abschnitt 4.6 (S. 56) diskutiert werden.

4.3.2 Der Ansatz von KRISTENSEN und ØSTERBYE

Konzeptuelle Modellierung

Die Ausgangsbasis für die Arbeiten über Rollenmodelle bildet der Ansatz der **konzeptuellen Modellierung** (*Conceptual Modeling*, [KØ94]). Die beiden zentralen Begriffe sind hierbei das **Phänomen** (*Phenomenon*) und das **Konzept** (*Concept*). Phänomene lassen sich (in der realen Welt) identifizieren, indem ihr Verhalten und wichtige Eigenschaften beobachtet werden. Konzepte dagegen existieren nur als Gedankenmodelle. Ein erstes Beispiel für Phänomene sind die (real existierenden) Hunde FIDO und ORLANDA. Durch einen Abstraktionsprozeß entsteht das Konzept *Hund*. Ein weiteres Beispiel ist das *Nachrichtenkonzept* der Programmierspache SMALLTALK. Phänomene wären in diesem Zusammenhang die während einer Programmausführung versendeten Nachrichten.

Der Modellbildungsprozeß ist in Abb. 11 (S. 28) dargestellt. Der erste Schritt besteht im Aufbau eines **konzeptuellen Verständnisses**. Hier ist eine Modellbildung für die Begriffe **Konzept**, **Phänomen** und **Abstraktion** vorzunehmen, um deren fundamentale Eigenschaften zu verstehen. Die **Konzeptgestaltung** umfaßt die Konstruktion eines konzeptuellen Modells für einen speziellen Anwendungsbereich durch die Identifikation der relevanten Phänomene und Konzepte. Die Spezifikation dieses Modells erfolgt häufig über eine informale Beschreibungssprache, z.B. auf der Basis einer grafischen Notation. Im nächsten Schritt kann eine auf dem konzeptuellen Verständnis basierende **Programmiersprache entworfen** werden. Beispielsweise unterstützen bei objektorientierten Sprachen die Mechanismen **Objekt** und **Klasse** die Modellierung von Phänomenen und Konzepten. Im letzten Schritt erfolgt die **Programmierung**. Das Programm selbst stellt eine Formalisierung der Konzeptgestaltung dar. Die (informale) Be-

[3] Denkbar wäre in diesem Fall, die Regelmenge als inkonsistent zu definieren.

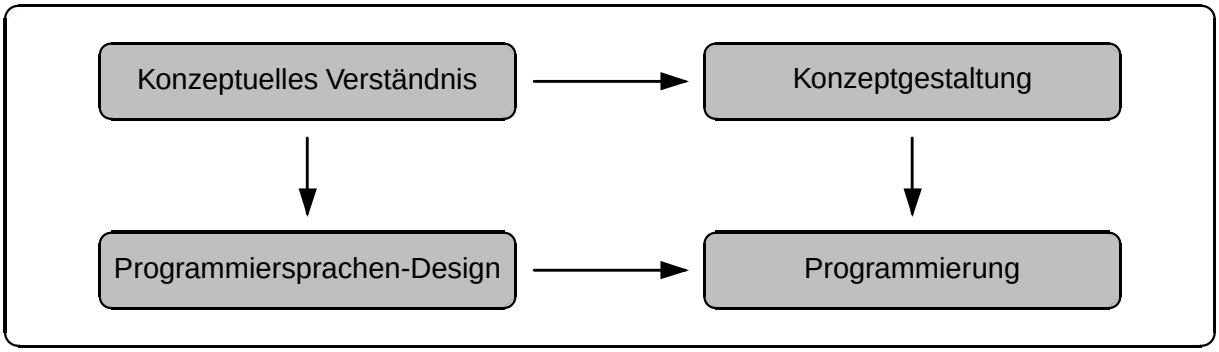


Abb. 11 Konzeptuelle Modellierung

schreibung wird in eine durch die Syntaxregeln der Programmiersprache vorgegebene formale Beschreibung umgesetzt.

Die semantische Lücke zwischen dem konzeptuellen Verständnis und den erzeugten Programmen wird umso kleiner sein, je besser das Programmiersprachen-Design auf die konzeptuelle Ebene zugeschnitten ist.

Konzepte und Abstraktionsprozesse

Ein Konzept setzt sich aus drei Bestandteilen zusammen:

- Die **Bezeichnung** (*Denomination*) des Konzepts.
- Die **Extension** des Konzepts. Dies ist die Menge der Phänomene, die sich dem Konzept zuordnen lassen.
- Die **Intention** des Konzepts. Hiermit ist die Menge der Eigenschaften gemeint, die die Phänomene aus der Konzeptextension gemeinsam haben.

Phänomene und Konzepte werden zur Modellbildung für einen Ausschnitt der Welt verwendet. Die dabei identifizierten Phänomene und Konzepte sind durch verschiedene Abstraktionsprozesse miteinander verbunden (vgl. [Wan89]): **Klassifikation** (*Classification*) und **Exemplifikation** (*Exemplification*), **Generalisierung** (*Generalization*) und **Spezialisierung** (*Specialization*) sowie **Aggregation** (*Aggregation*) und **Dekomposition** (*Decomposition*).

Die Abstraktionsprozesse können als Abbildungen zwischen den Mengen P, C und S aufgefaßt werden:

- P: Die Menge aller Phänomene.
- C: Die Menge der Konzepte.
- S: Die Potenzmenge von P.

Damit lassen sich die Abstraktionsprozesse wie folgt beschreiben:

- Klassifikation : $P \rightarrow C$
  Ein konkretes Phänomen wird einem Konzept zugeordnet. Damit gehört das Phänomen zur Extension dieses Konzepts. Beispielsweise können die Phänomene ORLANDA und FIDO als Hunde klassifiziert werden.
- Exemplifikation : $C \rightarrow P$
  Für ein bestimmtes Konzept wird ein Beispiel aus seiner Extension ausgewählt. FIDO könnte

als (typisches) Beispiel für das Konzept Hund dienen.

- Aggregation : $S \rightarrow C$
  Durch die Aggregation wird ein neues, komplexeres Konzept auf der Basis mehrerer, einfacherer Konzepte gebildet. Das Konzept Hund kann aus den Konzepten Kopf, Schwanz, Rumpf und Bein aggregiert werden.
- Dekomposition : $C \rightarrow S$
  Ein komplexes Konzept wird in einfachere Konzepte zerlegt. Ein Hund kann in die Bestandteile Kopf, Schwanz, Rumpf und Bein zerlegt werden.
- Spezialisierung : $C \rightarrow C$
  Ein zu einem Konzept C_1 eng verwandtes Konzept C_2 wird durch die Hinzunahme weiterer Eigenschaften gebildet. Damit ist die Menge der Phänomene, die in der Extension von C_2 liegen, nur noch eine Teilmenge aus der Extension von C_1. Das Konzept Hund kann zu Wachhund oder auch Jagdhund spezialisiert werden. Nicht jeder Jagdhund ist auch ein Wachhund und nicht jeder Hund ist ein Jagdhund.
- Generalisierung : $C \rightarrow C$
  Ein zu einem Konzept C_3 eng verwandtes Konzept C_4 wird durch das Weglassen von Eigenschaften erzeugt. Daher gehören alle Phänomene aus der Extension von C_3 automatisch zur Extension von C_4. In der Regel wird es aber noch zusätzliche Phänomene in der Extension des verallgemeinerten Konzepts C_4 geben. Beispielsweise kann das Konzept Hund zum Konzept Tier verallgemeinert werden.

Der Rollenbegriff als konzeptuelle Abstraktion

Die konzeptuelle Modellierung wird von KRISTENSEN zum einem verwendet, um objektorientierte Programmiersprachen zu vergleichen ([KØ96c]). Eine weitere Anwendung besteht in der Beschreibung und Analyse eines allgemeinen Modells für ein Rollenkonzept ([KØ96a]).

Rollen werden als eine Spezialisierung des allgemeinen Konzeptbegriffs eingeführt: Eine Rolle besitzt einen Namen (z.B. Mutter ist eine Rolle von Person), sie besitzt eine Extension (alle Frauen, die mindestens ein Kind haben) und eine Intention (die Eigenschaften, die zusätzlich erforderlich sind, damit eine Person eine Mutter ist). Die Unterscheidung zu einem allgemeinen Konzept liegt in der Frage der Individualität. Eine Rolle kann nicht eigenständig auftreten, es muß immer ein Phänomen geben, dem die Rolle zugeordnet ist und das weitere Eigenschaften neben denen der Rolle besitzt.

Betrachtet man die Rollenbildung als einen Abstraktionsprozeß[4], so stellt sich die Frage, inwieweit sie sich von den Abstraktionsprozessen Klassifikation und Spezialisierung unterscheidet. Die Rollenbildung stellt eine Form der Spezialisierung dar, weil durch die Übernahme einer Rolle neue Eigenschaften nutzbar werden. Auch stellt die Extension der Rolle eine Teilmenge der Extension des Grundkonzepts dar, welchem die Rolle hinzugefügt werden kann, da nicht alle Phänomene des Grundkonzepts notwendigerweise die Rolle übernehmen müssen. Ebenso findet über die Rollenbildung eine Klassifikation statt. Allerdings folgt aus der Tatsache, daß ein Phänomen einer Rolle zugeordnet wird, daß es auch einem bestimmten Grundkonzept zugehörig sein muß. Es wird somit neben einer bereits bestehenden Klassifikation eine zusätzliche

[4] In [KØ96a] wird dieser Abstraktionsprozeß mit dem englischen Kunstwort **Roleification** bezeichnet; hier wird stattdessen der Begriff **Rollenbildung** verwendet.

Klassifikation vorgenommen, die auch als **Rollenklassifikation** (*Role-Classification*) bezeichnet wird. Weitere wichtige Unterschiede der Rollenbildung im Vergleich zur Klassifikation und Spezialisierung sind:

- **Dynamische** Klassifikation:
  Ein Phänomen besitzt nur für einen Teil seiner Existenz eine bestimmte Rolle. Somit gehört dieses Phänomen auch nur zeitweise zur Extension der Rolle.
- **Keine** Generalisierung (im Sinne einer inversen Spezialisierung):
  Eine Person ist z.B. keine natürliche Generalisierung der Rolle Vater, obwohl die Vaterrolle als Spezialisierung von Person aufgefaßt werden kann.
- **Identität**:
  Die Identität einer Rolle ist immer an die Identität des zugehörigen Phänomens gekoppelt. Die Identität eines Vaters ist diejenige der zugehörigen Person.
- **Nur Funktionalitätserweiterung**:
  Durch die Übernahme einer Rolle erhält ein Phänomen neue Eigenschaften. Bereits bestehende Eigenschaften werden durch die Rolle nicht verändert. Die private Telefonnummer einer Person bleibt erhalten, wenn diese z.B. die Rolle Angestellter übernimmt.

Zusammenfassend wird in [KØ96a] festgestellt, daß die Rollenbildung einen eigenständigen Abstraktionsprozeß neben der Spezialisierung und der Klassifikation darstellt, welcher aber enge Beziehungen zu diesen beiden Abstraktionsprozessen besitzt.

Das Rollenkonzept – Terminologie und grafische Beschreibung

Für die Spezifikation des Rollenkonzepts wird die Terminologie und grafische Beschreibung aus [KØ96a] verwendet[5]. Da die Betrachtung des Rollenkonzepts hier im objektorientierten Umfeld stattfindet, werden statt der allgemeineren Begriffe Phänomen und Konzept alternativ auch die dort gebräuchlicheren Begriffe verwendet, die allerdings für den Rollenkontext angepaßt bzw. ergänzt wurden:

- **Objekte** sind Einheiten mit einer Identität, die unabhängig von Rollen existieren können. Sie werden auch als **intrinsische** (*intrinsic*) Objekte bezeichnet.
- Objekte werden durch (intrinsische) **Klassen** beschrieben.
- Eine **Rolle** ist die Beschreibung für die zugehörigen **Rolleninstanzen**. Eine Rolleninstanz hat einen Zustand und ein Verhalten, das über die Methoden spezifiziert wird.
- Seien obj ein intrinsisches Objekt und r_1 bis r_n die Menge der aktuell gebundenen Rolleninstanzen. Jede Menge, die obj und mindestens eine Rolle r_i enthält, wird als Subjekt bezeichnet. Das intrinsische Objekt zusammen mit allen seinen gebundenen Rollen bildet das **abgeschlossene Subjekt** (*Closured Subject*).

Abb. 12 (S. 31) zeigt die Darstellung einer Klasse C mit den zugeordneten Rollen R1 und R2. Eine **Eigenschaft** (*Property*) einer Klasse bzw. einer Rolle ist typischerweise ein Attribut oder eine Methode.[6]

[5] In anderen Veröffentlichungen von KRISTENSEN wird eine leicht veränderte grafische Beschreibung benutzt (vgl. z.B. [Kri95]).

[6] In der Analysephase sind Attribute noch sichtbar, da sie unter anderem zur Identifikation der Klassen (bzw. hier

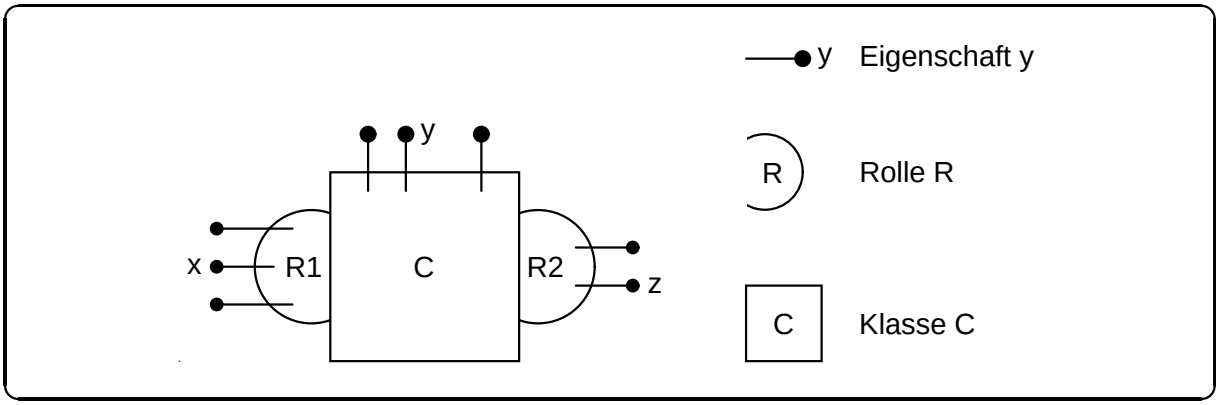


Abb. 12 Die Darstellung von Klassen und Rollen

Rollen können neben Klassen auch anderen Rollen zugeordnet werden. Beispielsweise kann eine Person die Professorenrolle übernehmen und danach für eine Konferenzorganisation (in seiner Rolle als Professor) die Rollen Gutachter und Vortragender ausüben. Abb. 13 zeigt die zugehörige Notation.

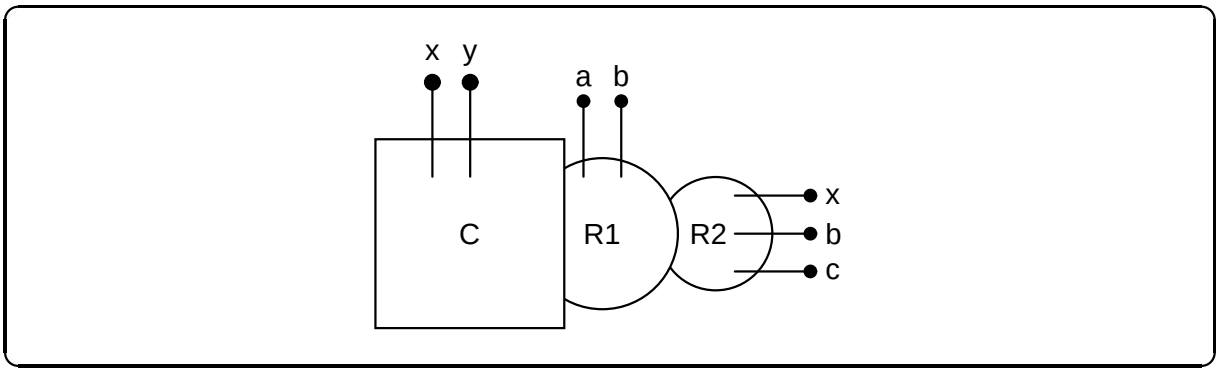


Abb. 13 Die Zuordnung von Rollen zu Rollen

Abstraktionsprozesse für Rollen

Rollen stellen eine Spezialisierung eines allgemeinen Konzepts dar. Somit stellt sich die Frage, welche Abstraktionsprozesse für das Rollenkonzept angewendet werden können:

- **Klassifikation**
  Werden einem Objekt der Klasse K Rolleninstanzen der Rollen R_1 bis R_n zugeordnet, so kann das entstehende Subjekt sowohl als K wie auch als R_i ($i \in \{1, \cdots, n\}$) klassifiziert werden.
- **Exemplifikation**
  In objektorientierten Programmiersprachen wird die Exemplifikation üblicherweise durch die Objektinstanziierung (Objekterzeugung) vorgenommen. Für Rollen gilt, daß eine isolierte Instanziierung nicht möglich ist. Es muß zusätzlich eine Objekt- oder Rolleninstanz vorliegen, an die die neue Rolleninstanz gebunden wird.

Konzepte) benötigt werden. In der Entwurfs- und Implementierungsphase sollten zur Umsetzung des Geheimniskonzepts ([Par72]) nur noch Operationen sichtbar sein.

- **Spezialisierung**
  Genauso wie Klassen können Rollen spezialisiert werden. In Abb. 14 stellen die Rollen R1 und R2 Spezialisierungen der Rolle R dar. Eine spezialisierte Rolle erbt alle Eigenschaften der allgemeineren Rolle und kann natürlich neue Eigenschaften definieren. Zusätzlich ist es möglich, daß geerbte Eigenschaften verändert werden. R2 erbt x, y, z, a und b, wobei x und b redefiniert werden. Die neue Eigenschaft von R2 ist c. Die spezialisierten Rolleninstanzen können zu denselben Objekten[7] gebunden werden, wie die Instanz der verallgemeinerten Rolle.
  Die beiden Möglichkeiten zur Anwendung der Vererbung im Zusammenhang mit dem Rollenkonzept lassen sich auch kombinieren, wie dies in Abb. 15 (S. 33) dargestellt wird. Eine R1-Rolleninstanz darf nur Objekten der Klasse CC zugeordnet werden, während eine R2-Rolleninstanz sowohl C- als auch CC-Objekten zur Verfügung steht, da die Klasse CC die Rolle R von C erbt.

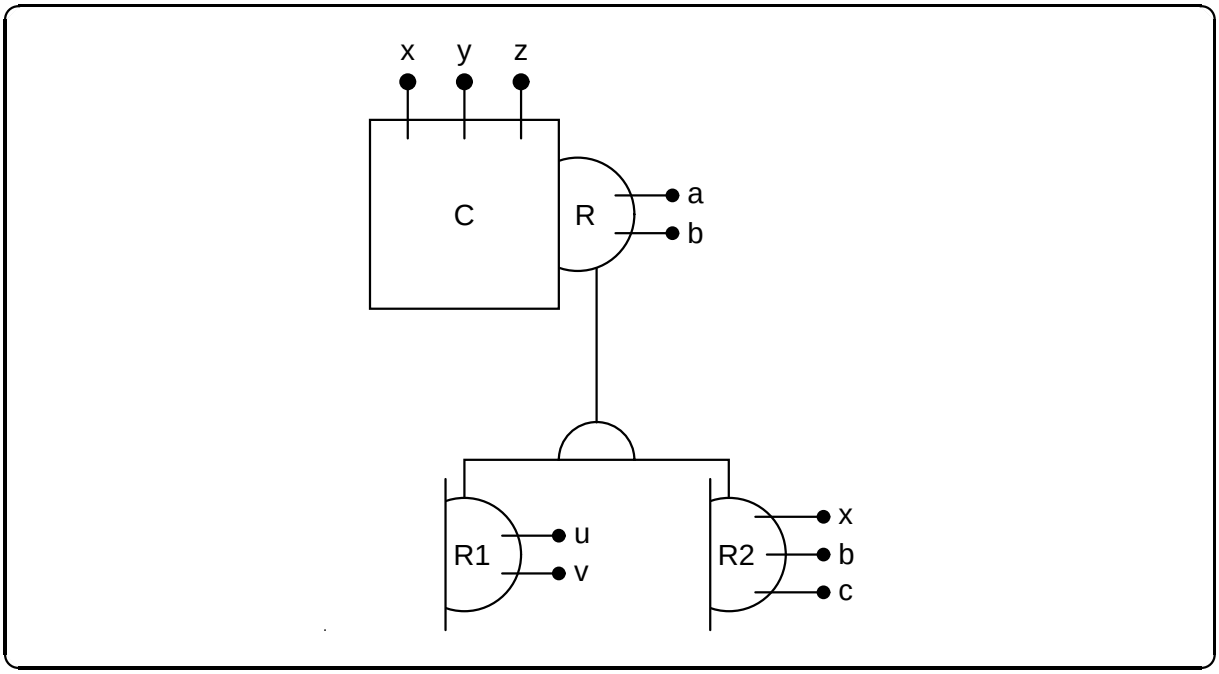


Abb. 14 Die Spezialisierung von Rollen

- **Generalisierung**
  Da eine Rolle R selbst wieder zu neuen Rollen R1 bis Rn spezialisiert werden kann, läßt sich R als Generalisierung dieser Unterrollen ansehen. Die intrinsische Klasse einer Rolle stellt dagegen nicht notwendigerweise eine natürliche Generalisierung dar (vgl. S. 30).
- **Aggregation**
  Eine Rolle kann aus bereits bestehenden Rollen zusammengesetzt werden. In Abb. 16 (S. 33) besteht R aus R1 und R2. Für die Eigenschaften der Aggregatrollen (R1 und R2) bestehen zwei Möglichkeiten: (1) Die Eigenschaft kann an der Schnittstelle der neuen Rolle sichtbar sein, d.h. sie wird **vererbt** (*hereditary*) (u aus R1 und b aus R2). (2) Die Eigenschaft wird **versteckt** (*hidden*). In Abb. 16 (S. 33) sind dies v aus R1 sowie x und c aus R2. Zu-

[7] Statt des Begriffs intrinsisches Objekt wird in der Folge nur noch der Begriff Objekt verwendet.

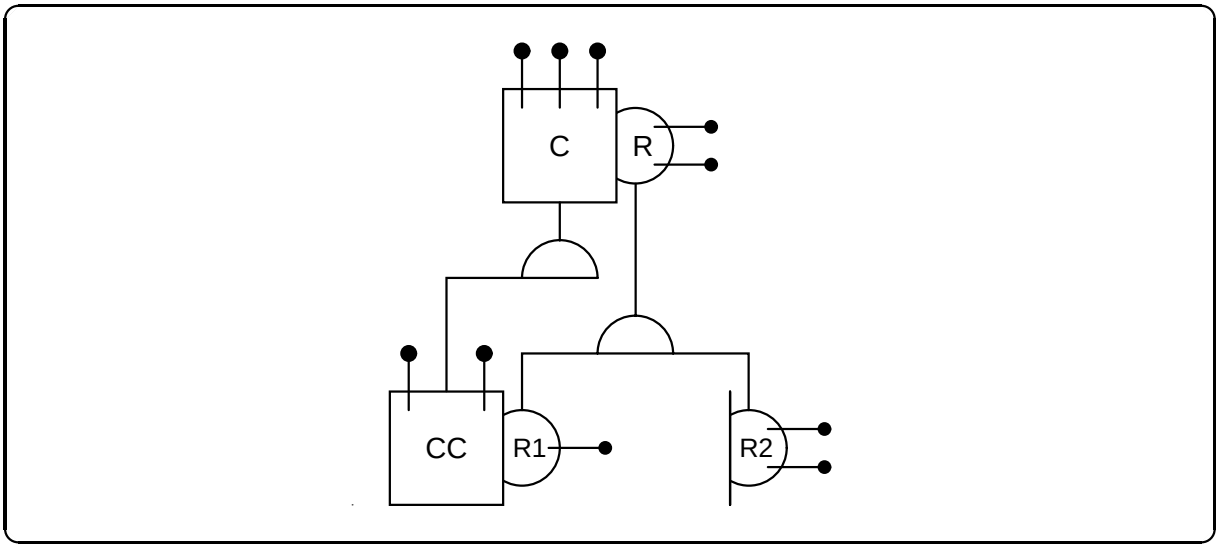


Abb. 15 Die Anwendung der Vererbungskonzepte für Klassen und Rollen

sätzlich kann die neue Rolle weitere Eigenschaften besitzen. Basiert eine neue Eigenschaft E auf Eigenschaften der Aggregatrollen, so wird E in [Kri95] als **emerging** bezeichnet. Die Aggregatrollen müssen zu derselben Klasse wie die aggregierte Rolle gehören, d.h. im Beispiel aus Abb. 16 sind sowohl R1 und R2 als auch R Rollen von C. Eine Instanz c von C, die eine Rolleninstanz r von R übernimmt, besitzt automatisch auch Rolleninstanzen r1 von R1 und r2 von R2. Damit kann auf c über drei Rolleninstanzen (r, r1 und r2) zugegriffen werden.[8]

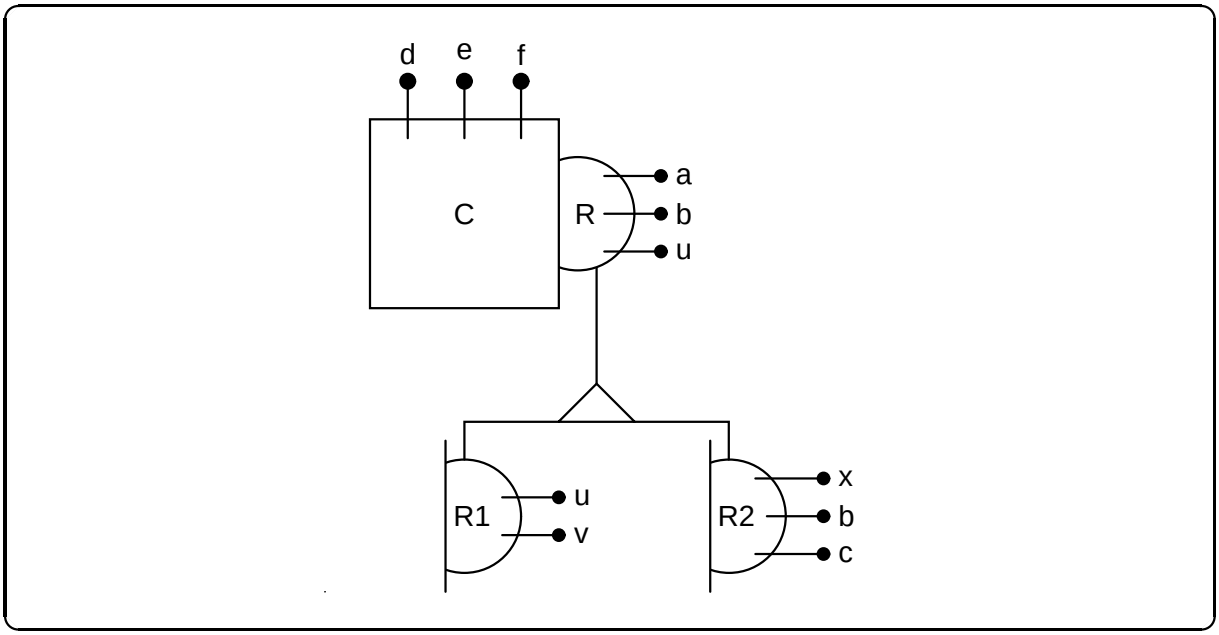


Abb. 16 Die Aggregation von Rollen

[8] Diese Aussage ist in [Kri95] bzw. [KØ96a] zwar nicht explizit erwähnt, sie läßt sich aber aus dem Kontext mit hoher Wahrscheinlichkeit ableiten.

- **Dekomposition**
  Eine Rolle an sich kann nicht das Ergebnis der Dekomposition eines Subjekts sein. Ein Subjekt (Objekt plus Rollen) läßt sich nicht in seine Rollen zerlegen, da die Rollen keine eigenständigen Bestandteile darstellen und eine Rolleninstanz ohne die zugehörige Objektinstanz nicht existieren kann. Allerdings kann eine Rolle, die selbst wieder aus Aggregatrollen besteht, in diese Aggregatrollen zerlegt werden, weil die Dekomposition die Umkehrung der Aggregation darstellt.

Zusammenfassend läßt sich festhalten, daß alle für allgemeine Konzepte definierten Abstraktionsprozesse sinnvoll auf das Rollenkonzept angewendet werden können.

Rollen für Eigenschaften

Eigenschaften (d.h. Attribute und Methoden) können ebenfalls wieder als spezielle Konzepte aufgefaßt werden. Daher ist es möglich, eine Rolle für eine Eigenschaft zu definieren, die **Eigenschaftsrolle** (*Property-Role*). Die Notation ist in Abb. 17 dargestellt: b ist eine Eigenschaftsrolle für y.

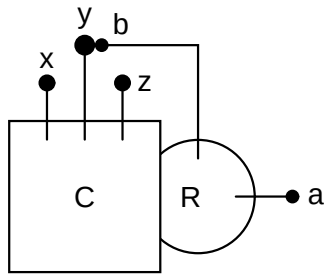


Abb. 17 Die Definition einer Eigenschaftsrolle

Bei der Festlegung der Semantik einer Eigenschaftsrolle ist zu unterscheiden, ob es sich um eine Attributrolle oder eine Methodenrolle handelt:

- Attributrolle
  Wird eine Referenz r des Rollentyps R verwendet, so liefert r.b die Kombination aus b und y zurück. Wenn beispielsweise C die Klasse Person beschreibt und y eine Liste von persönlichen Daten sowie R die Rolle Tennisspieler und b die Liste seiner Erfolge, dann stellt r.b eine Referenz auf die persönlichen Daten ergänzt um die Tenniserfolge dar. Ist b dagegen keine Attributrolle, sondern nur ein Attribut von R, dann ermöglicht r.b nur den Zugriff auf die Erfolge der Person in seiner Rolle als Tennisspieler.
- Methodenrolle
  Sind in dem Beispiel aus Abb. 17 y und b Methoden, so hat der Aufruf r.b die Ausführung beider Methoden zur Folge. Allerdings wird nicht näher spezifiziert, in welcher Reihenfolge die beiden Methoden auszuführen sind.

Die Subjektabstraktion

Ein Subjekt wurde als Objekt mit Rolleninstanzen definiert (vgl. S. 30). Das Objekt modelliert zusammen mit seinen Rolleninstanzen wieder ein Phänomen. Daher wird **Subjekt** (*Subject*) als weiteres Konzept eingeführt. Abb. 18 zeigt die Notation für Subjekte. CC entsteht durch die Abstraktion der Klasse C mit den Rollen R1 und R2 zu einem Subjekt. Da Subjekt jetzt die Benennung für ein Konzept darstellt, wird ein Objekt zusammen mit den zugehörigen Rollen als Subjektinstanz bezeichnet.[9]

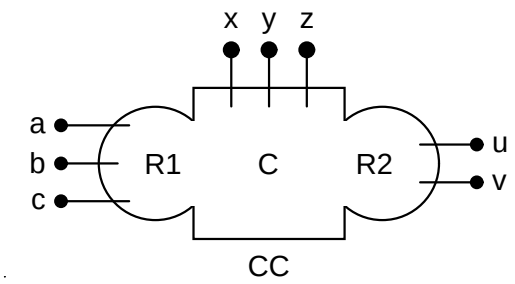


Abb. 18 Die Subjektabstraktion

Für eine Subjektinstanz cc des Typs CC sind vier Klassifikationen möglich:[10]

- Referenz des Typs C ⇒ Zugriff auf x, y und z
- Referenz des Typs R1 ⇒ Zugriff auf a, b, c, x, y und z
- Referenz des Typs R2 ⇒ Zugriff auf u, v, x, y und z
- Referenz des Typs CC ⇒ Zugriff auf a, b, c, u, v, x, y und z

Durch die Einführung der Subjektabstraktion ist zu klären, in welcher Form Methodenrollen auszuführen sind.[11] Als Beispiel soll die in Abb. 19 (S. 36) dargestellte Struktur vorliegen: Eine Subjektinstanz cc des Typs CC besitzt (neben den Rolleninstanzen r2 und r3) eine Instanz r1 der Rolle R1. Alle drei Rollen definieren Methodenrollen für die Methode y.

Für die Zugriffe über cc auf die Methoden des Objekts sind verschiedene Fragen zu klären:

- Besitzen die Zugriffe cc.b, cc.c und cc.y diesselbe Semantik?
- Welche Methodenrollen werden aktiviert?
- In welcher Reihenfolge werden y und die Methodenrollen ausgeführt?

Auf die erste Frage wird in [KØ96a] nicht näher eingegangen. Für die zweite Frage werden zwei Ansätze vorgestellt:

- Der Zugriff cc.y aktiviert zusätzlich die Methodenrollen b und c der Subjektinstanz.
- Ein Aufruf von y aktiviert alle Methodenrollen, also neben b und c auch a, d.h. es wird das abgeschlossene Subjekt als Grundlage verwendet (Closured-Subject-Ansatz).

[9] Die etwas inkonsequente Begriffsbildung für Subjekt wurde aus [KØ96a] übernommen.

[10] Es wird davon ausgegangen, daß cc aus dem Objekt c und den Rolleninstanzen r1 und r2 besteht; andernfalls wäre es keine Subjektinstanz von CC.

[11] Auf Attributrollen wird in [KØ96a] und [Kri95] nicht näher eingegangen.

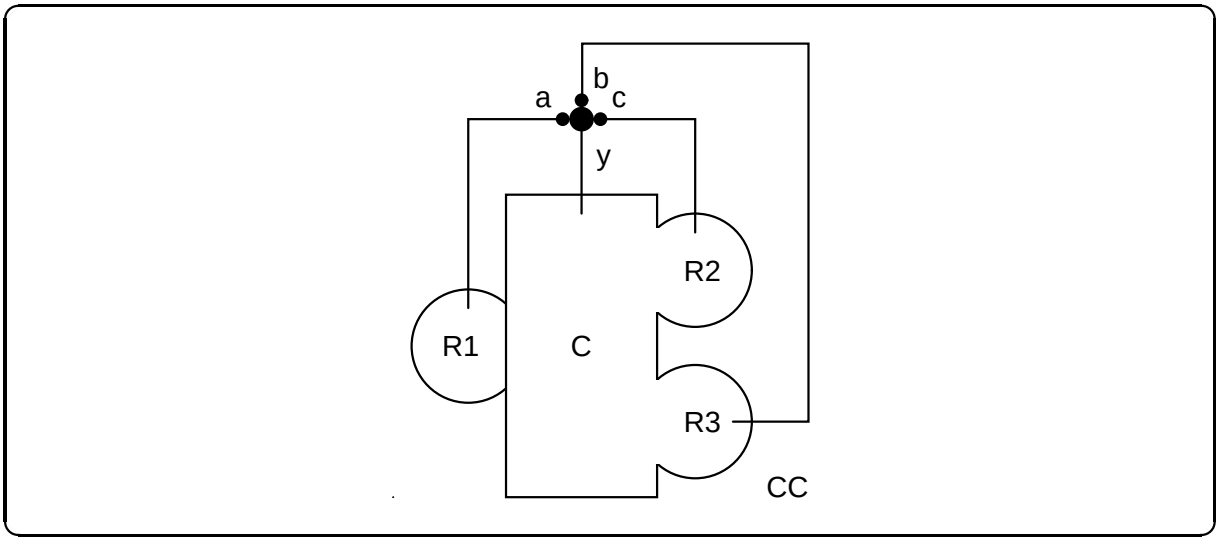


Abb. 19 Subjekte und Methodenrollen

Eine Lösung für die dritte Frage ist, zunächst alle Methodenrollen in einer nicht näher spezifizierten Reihenfolge auszuführen (hier b und c). Jede Methodenrolle übergibt zu einem bestimmten Zeitpunkt die Kontrolle an die Objektmethode, allerdings wird die Objektmethode erst aktiviert, wenn alle Methodenrollen diesen Punkt erreicht haben. Ist die Objektmethode fertig bearbeitet, erfolgt die Rückkehr zu den Methodenrollen. Damit umhüllen die Methodenrollen die Objektmethode.

Zeitrestriktionen für Rollen

Die wesentliche Grundlage des Rollenkonzepts besteht in der dynamischen Übernahme und Abgabe von Rollen durch ein Subjekt (bzw. Objekt). Neben der Definition, **welche** Rollen ein Subjekt annehmen kann, sollte auch spezifizierbar sein, in welcher **Reihenfolge** diese Rollen an das Subjekt gebunden werden können. In [KØ96a] wird hierfür eine grafische Notation in Diagrammform verwendet, die in Abb. 20 (S. 37) dargestellt ist.

Für ein Diagramm muß dann jeweils angegeben werden, welche Klasse bzw. welche Rolle der Ausgangspunkt für das Binden der Rollen ist.[12] Die Notation besteht aus den folgenden Grundstrukturen, die dann beliebig kombinierbar sind und festlegen, welche Rollen in welcher Reihenfolge an ein Objekt oder wieder eine Rolle gebunden werden können.

- **Atomare Box**
  Der in der Box angegebene Rollenname sagt aus, daß eine Rolleninstanz dieses Rollentyps gebunden ist.
- **Sequenz**
  Die Boxen werden zeitlich hintereinander ausgeführt[13]. Ein durchgezogener Pfeil sagt dabei aus, daß zwischen der Ausführung der beiden Boxen keine zeitlich Lücke bestehen darf, während bei einem gestrichelten Pfeil die erste Box endet und erst eine Zeit verstreichen

[12] Eine Rolle kann ja selbst wieder Rollen besitzen.

[13] Die Ausführung einer Box soll in diesem Zusammenhang bedeuten, daß zu Beginn die entsprechenden Rollen gebunden werden und am Ende die Rollen abgegeben werden.

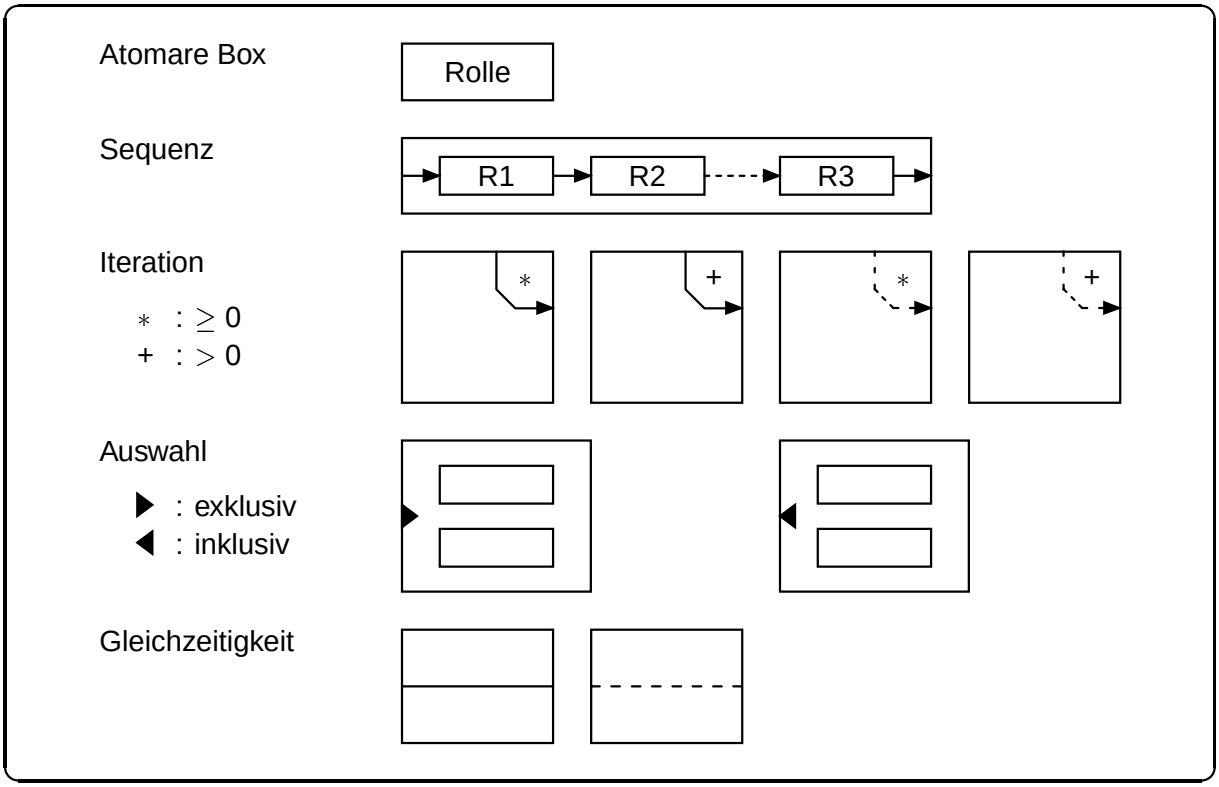


Abb. 20 Die Spezifikation von Zeitrestriktionen

kann, bevor die nächste Box beginnt. Für das Beispiel aus Abb. 20 gilt: Zunächst wird die Rolle R1 übernommen. Danach erfolgt ein direkter Übergang von R1 nach R2, d.h. R1 wird aufgegeben und gleichzeitig R2 übernommen. Nach dem Löschen von R2 darf eine gewisse Zeit vergehen, bevor die Rolle R3 angenommen wird.

- **Iteration**
  Die Box wird mehrmals ausgeführt. Ein gestrichelter Pfeil sagt hierbei wieder aus, daß zwei Ausführungen nicht direkt hintereinander erfolgen müssen.
- **Exklusive Auswahl**
  Genau eine der angegebenen Boxen darf ausgeführt werden.
- **Inklusive Auswahl**
  Mindestens eine Box muß ausgeführt werden.
- **Gleichzeitigkeit**
  Boxen lassen sich übereinander stapeln. Werden zwei Boxen mit einer durchgezogenen Linie voneinander getrennt, dann müssen ihre Ausführungen gleichzeitig beginnen und enden. Sind sie dagegen durch eine gestrichelte Linie getrennt, so reicht eine zeitliche Überlappung in ihren Ausführungen aus.

Umsetzung des Rollenkonzepts

In [KØ96a] werden Umsetzungen des vorgestellten Rollenkonzepts mit den Programmiersprachen BETA ([MMN93]) und SMALLTALK ([GR89]) beschrieben. Die Implementierungen werden als experimentell bezeichnet, da sie keine vollständigen Realisierungen darstellen.

In BETA wurde eine Spracherweiterung vorgenommen, um Rollen und die Spezialisierung von Rollen zu beschreiben. Die SMALLTALK-Implementierung basiert auf einer Klasse `Role`, die die Oberklasse für alle Rollenklassen darstellt. Da SMALLTALK nur Einfachvererbung unterstützt, ist es nicht möglich, daß eine Rollenklasse eine Nicht-Rollenklasse als Oberklasse besitzt.

Da eine Rolleninstanz immer an ein Objekt gebunden sein muß, wurde in BETA hierfür ein neues Sprachkonstrukt eingeführt. In SMALLTALK wurde die Klasse `Object` um die Methode `bindRole` erweitert; diese besitzt als Parameter eine Rollenklasse und bindet eine Rolleninstanz an das Objekt. Als Ergebnis liefert ihr Aufruf die Referenz auf die Rolleninstanz. Über die Referenzen auf die Rolleninstanzen sind die unterschiedlichen Sichtweisen auf ein Objekt realisiert. Bei beiden Implementierungen ist der Zugriff auf die Eigenschaften des (intrinsischen) Objekts über die Rolleninstanzreferenz möglich.
Zur Realisierung von Subjektreferenzen wurde das neue BETA-Sprachkonstrukt zum Binden von Rollen derart erweitert, daß mehrere Rolleninstanzen einer Referenz zugeordnet werden können. Der SMALLTALK-Ansatz stellt hierfür eine neue Klasse `SubjectRef` zur Verfügung. Ein `SubjectRef`-Objekt enthält die Referenzen auf die Rolleninstanzen und leitet Nachrichten an die entsprechenden Rolleninstanzen weiter.

Für die Implementierung der Zeitrestriktionen wird neben dem Binden von Rollen das Auflösen der Bindung einer Rolle (`unbind`-Operation) benötigt. Erfolgt ein explizites Auflösen der Rolleninstanzbindung über eine `delete`-Operation, so entsteht das Problem der **hängenden Referenzen** (*Dangling Reference*). Über derartige Referenzen ist der Zugriff auf das intrinsische Objekt noch möglich, obwohl die Rolleninstanz nicht mehr existiert.
Erlaubt man stattdessen das Entfernen (`remove`-Operation) einer Rolleninstanz von seinem intrinsischen Objekt, dann treten **hängende Rolleninstanzen** auf. Auf die Rolleninstanz kann noch zugegriffen werden, es ist aber kein intrinsisches Objekt mehr zugeordnet. Um dieses Problem zu umgehen, wurden sowohl im BETA- als auch im SMALLTALK-Ansatz *Ghost*-Objekte eingeführt, die eine Rolleninstanz nach der Auflösung der Bindung zu ihrem intrinsischen Objekt übernehmen. Eine Rolleninstanz kann dann intern abfragen, ob sie an ein Ghost-Objekt gebunden ist.[14] Wird für eine Rolleninstanz die letzte Referenz gelöscht, so wird diese automatisch dem Garbage Collector übergeben. Auf diesem Weg erfolgt ein implizites Löschen der Rolleninstanz.
Als eine weitere Operation wurde während des Implementierungsprozesses das **Transferieren** von Rolleninstanzen (*Role Transfer*) identifiziert. Im einfachsten Fall entspricht dieses dem Austausch des intrinsischen Objekts. Ein Anwendungsbeispiel wäre die Neuzuordnung der Rolle eines Bürgermeisters zu einer Person nach einer Wahl.
Auf eine Integration der modellierbaren Zeitrestriktionen (vgl. Abb. 20 (S. 37)) in die Implementierung des Rollenkonzepts wurde verzichtet. Die Kontrolle der Zeitrestriktionen bleibt damit dem Anwendungsprogrammierer überlassen.

Für die Realisierung der Methodenrollen wurden zwei unterschiedliche Ansätze verfolgt. Die BETA-Implementierung unterstützt nur die Aktivierung von Methodenrollen über eine Subjektreferenz, wobei die Methodenrollen die Objektmethode umhüllen (vgl. S. 36). In der SMALLTALK-Implementierung wurde der Closured-Subject-Ansatz verfolgt. Die prinzipielle Ausführungsreihenfolge der Methodenrollen und der Objektmethode entspricht dem BETA-Ansatz.

[14] Wie eine Rolleninstanz, die an ein Ghost-Objekt gebunden ist, auf Operationsaufrufe reagiert, ist in [KØ96a] allerdings nicht näher beschrieben.

Die Bindung zwischen den Methodenrollen und der Objektmethode wird hier über eine Namenskonvention erreicht.[15] Für eine Objektmethode `name` werden in einer Rolle jeweils zwei Methodenrollen definiert: `preName` und `postName`. Ein Aufruf von `name` aktiviert dann zunächst alle `preName` Methodenrollen, danach wird `name` ausgeführt und zum Schluß alle `postName` Methodenrollen.

Resultierende Anforderungen an ein Rollenkonzept

In [Kri95] werden 6 wesentliche Eigenschaften für Rollenkonzepte identifiziert:

- C1: **Sichtbarkeit** (*Visibility*): Die Sicht auf ein Objekt und damit auch die Zugriffsmöglichkeiten werden über eine Rolleninstanz eingeschränkt. Außer den Methoden der Rolleninstanz sind nur die Methoden des Objekts aufrufbar. Rollen können daher für eine Mehrfachklassifikation von Objekten verwendet werden.
- C2: **Abhängigkeit** (*Dependency*): Eine Rolleninstanz kann nicht ohne ein Objekt existieren. Die Methoden einer Rolleninstanz können auf die Objektmethoden zugreifen, die Objektmethoden aber nicht auf die Rolleninstanzmethoden.
- C3: **Identität** (*Identity*): Das Objekt besitzt zusammen mit seinen Rolleninstanzen eine (gemeinsame) Identität.[16]
- C4: **Dynamik** (*Dynamicity*): Rolleninstanzen lassen sich während der Lebenszeit eines Objekts hinzufügen und entfernen.
- C5: **Multiplizität** (*Multiplicity*): Für dasselbe Objekt können mehrere Instanzen derselben Rolle existieren.
- C6: **Abstrahierbarkeit** (*Abstractivity*): Rollen können zur Klassifikation eingesetzt werden und lassen sich in Generalisierungs- und Aggregationshierarchien anordnen.

Die Eigenschaft C5 ist im Vergleich zu [KØ96a] hinzugekommen.

Simulation des Rollenkonzepts

In [Kri95] wird untersucht, inwieweit eine Simulation des Rollenkonzepts durch die Konzepte **Spezialisierung**, **Aggregation** und **Assoziation** möglich ist. Die Ergebnisse sind in Tab. 1 (S. 40) zusammengefaßt.

Anwendungen der konzeptuellen Modellierung mit Rollen

In [BK99] wird am Beispiel der Konstruktion eines Schiffes demonstriert, wie die Rollenmodellierung in einem großen Projekt einsetzbar ist. Klassen und Rollen werden verwendet, um zunächst unterschiedliche Sichtweisen des Projekts zu beschreiben (z.B. die technische Konstruktion, der Zusammenbau des Schiffes, die Logistikseite und die wirtschaftliche Betrachtung). Jede Sichtweise definiert eine eigene Hierarchie aus Klassen und Rollen. Diese Hierarchien werden dann in ein aggregiertes Modell überführt.

[15] Wie die Bindung im BETA-Ansatz stattfindet, wird in [KØ96a] nicht näher beschrieben.

[16] Die Identität darf nicht mit den Referenzen auf die Rolleninstanzen verwechselt werden. Diese ermöglichen die unterschiedlichen Sichtweisen auf dasselbe Objekt.

	Spezialisierung	Aggregation	Assoziation
Sichtbarkeit	–	(+)	(–)
Abhängigkeit	+	–	(+)
Identität	+	–	–
Dynamik	(–)	(+)	+
Multiplizität	–	+	+
Abstrahierbarkeit	(+)	+	+

Tab. 1 Probleme mit der Simulation von Rollen

Eine weitere Anwendung besteht in der Komposition von Subjekten unter Verwendung von Rollen ([Kri97]). Über den Subjektbegriff läßt sich modellieren, daß es verschiedene Sichtweisen auf ein Objekt gibt ([HO93]). Betrachtet man nun ein größeres Anwendungssystem, dann liefert der Analyseprozeß zunächst mehrere Teilmodelle, die jeweils unterschiedliche Teilbereiche des Gesamtmodells beschreiben. Jedes Teilmodell enthält ein oder mehrere Subjektdefinitionen. Verwendet man für die Modellierung von Subjekten das Rollenkonzept, so können in unterschiedlichen Teilmodellen die gleichen Rollen auftreten. Um ein korrektes Gesamtsystemmodell zu erhalten, ist es in diesem Fall notwendig, eine Komposition der zugehörigen Subjekte vorzunehmen. In [Kri97] wird diskutiert, wie dieser Kompositionsprozeß unter Anwendung der konzeptuellen Rollenmodellierung stattfinden kann.

Die Modellierung von Aktivitäten (*Activity*) wird in [KM96] beschrieben. Eine Aktivität wird dabei als ein Abstraktionsmechanismus für die Modellierung des Zusammenspiels von Objekten aufgefaßt. Auch hier stellt das Rollenkonzept wieder eine wichtige Grundlage dar. Auf weitere Ansätze zur Beschreibung von Objektkollaborationen wird in Abschnitt 4.4 (S. 42) eingegangen. Da auch bei Entwurfsmustern ([GHJV95]) die Zusammenarbeit der beteiligten Objekte einen zentralen Aspekt darstellt, lassen sich viele Muster unter Verwendung des Rollenmodellkonzepts prägnanter formulieren. In [KO96b] und [Rie96] werden zahlreiche Muster neu formuliert.

Bewertung

Die konzeptuelle Modellierung stellt sicherlich das am weitesten ausgearbeitete Rollenkonzept der Analysephase dar. Als Nachteile dieses Modellierungsansatzes sind festzuhalten:

- Eine Mehrfachvererbung ist für Rollen nicht vorgesehen.
- Die Komposition von Methodenrollen wird nicht eindeutig spezifiziert.
- Für die grafische Beschreibung der Zeitrestriktionen wurde nicht untersucht, welche Mächtigkeit dieser Beschreibungsmechanismus besitzt, d.h. welchen Umfang die Menge der konkret spezifizierbaren Zeitrestriktionen hat.
- Es wird nicht näher darauf eingegangen, wie mehrere Rolleninstanzen desselben Rollentyps unterschieden werden. Insgesamt werden Mehrfachinstanzen eines Rollentyps nur am Rande erwähnt. Es wird nicht analysiert, ob sich diese nahtlos in die vorgestellten Konzepte integrieren lassen.

Die Implementierungen wurden nur experimentell durchgeführt. Eine vollständige Umsetzung der vorgestellten Konzepte liegt daher nicht vor. Außerdem wurde auf eine Realisierung der Zeitrestriktionen vollständig verzichtet. Diese bleibt dem Anwendungsprogrammierer überlassen. Dem Ansatz aus Abb. 11 (S. 28) entsprechend wäre es eigentlich konsequent gewesen, eine eigene Programmiersprache zu entwerfen, die alle notwendigen Sprachkonstrukte zur Umsetzung des vorgestellten Rollenkonzepts enthält. Damit wäre die semantische Lücke zwischen dem konzeptuellen Verständnis und der Programmerstellung geschlossen oder zumindest reduziert worden.[17]

[17] Ob sich eine derartige Programmiersprache dann am *Markt* durchsetzt, ist natürlich eine andere Frage.

4.4 Rollenkonzepte zur Modellierung von Objektkollaborationen

4.4.1 Die OOram-Methode

Die OORAM-Methode[18] ([RWL95][19], [Ree97]) ist aus dem OORASS-Ansatz[20] hervorgegangen ([RAB+92], [WBJ90]). OORAM dient zur Entwicklung großer, komplexer Systeme und ist von seinem Einsatzgebiet her vergleichbar mit Methoden wie OMT[21] ([RBP+91]), der BOOCH-Methode ([Boo91]) oder OOSE[22] ([JCJÖ92]). Der Systementwicklungsprozeß wird aus drei Blickwinkeln betrachtet:

- Technologieorientiert
- Prozeßorientiert
- Organisatorisch

Die Anwendung von Rollenmodellen gehört zur technologieorientierten Sichtweise. In einem ersten Analyseschritt werden abgegrenzte Bereiche der realen Welt (*Area of Concern*) identifiziert. Für jeden dieser Zuständigkeitsbereiche wird ein **Rollenmodell** erstellt. Ein Rollenmodell abstrahiert die Interaktionsmuster der beteiligten Objekte in ein Interaktionsmuster von Rollen. Von den Objekten werden nur noch diejenigen Eigenschaften betrachtet, die für die Beschreibung und das Verständnis des gewählten Zuständigkeitsbereichs notwendig sind. Die wesentlichen Eigenschaften einer Rolle sind ([RWL01, S. 39]):

- Eine Rolle beschreibt eine Menge von Objekten, die in einem Interaktionsmuster dieselbe Position einnehmen.
- Eine Rolle besitzt Attribute und reagiert auf Nachrichten mit der Aktivierung der entsprechenden Operationen.
- Eine Rolle besitzt eine eigenständige Identität.
- Die Eigenschaften einer Rolle stellen eine Teilmenge der Objekteigenschaften dar ([RWL01, S. 87]).
- Ein Objekt kann mehrere Rollen aus demselben oder verschiedenen Rollenmodellen annehmen ([RWL01, S. 88]).
- Eine Rolle kann von mehreren Objekten übernommen werden ([RWL01, S. 88]).

Das Rollenmodell, zu dem eine Rolle gehört, legt zusätzlich fest, an welche anderen Rollen Nachrichten gesendet werden dürfen.

Die OORAM-Methode definiert insgesamt 10 unterschiedliche Sichtweisen für Rollenmodelle ([RWL01, S. 89]). Für das Zusammenspiel der Rollen ist die **Kollaborationssicht** (*Collaboration View*) am wichtigsten. Abb. 21 (S. 43) zeigt die Notation der Kollaborationssicht. Das obere Rollenmodell (Frontloading Modell) spezifiziert, welche Rollen ein Auftragobjekt annimmt, um den Frontloading-Algorithmus umzusetzen. Dieser Algorithmus soll ermitteln, zu welchem Zeit-

[18] OORAM: Object-Oriented Role Analysis and Modeling

[19] Das Buch selbst ist nicht mehr im Handel erhältlich, eine Version des zugehörigen Manuskripts kann von `http://folk.uio.no/trygver/documents/book11d.pdf` geladen werden ([RWL01]).

[20] OORASS: Object-Oriented Role Analysis, Synthesis and Structuring

[21] OMT: Object Modeling Technique

[22] OOSE: Object-Oriented Software Engineering

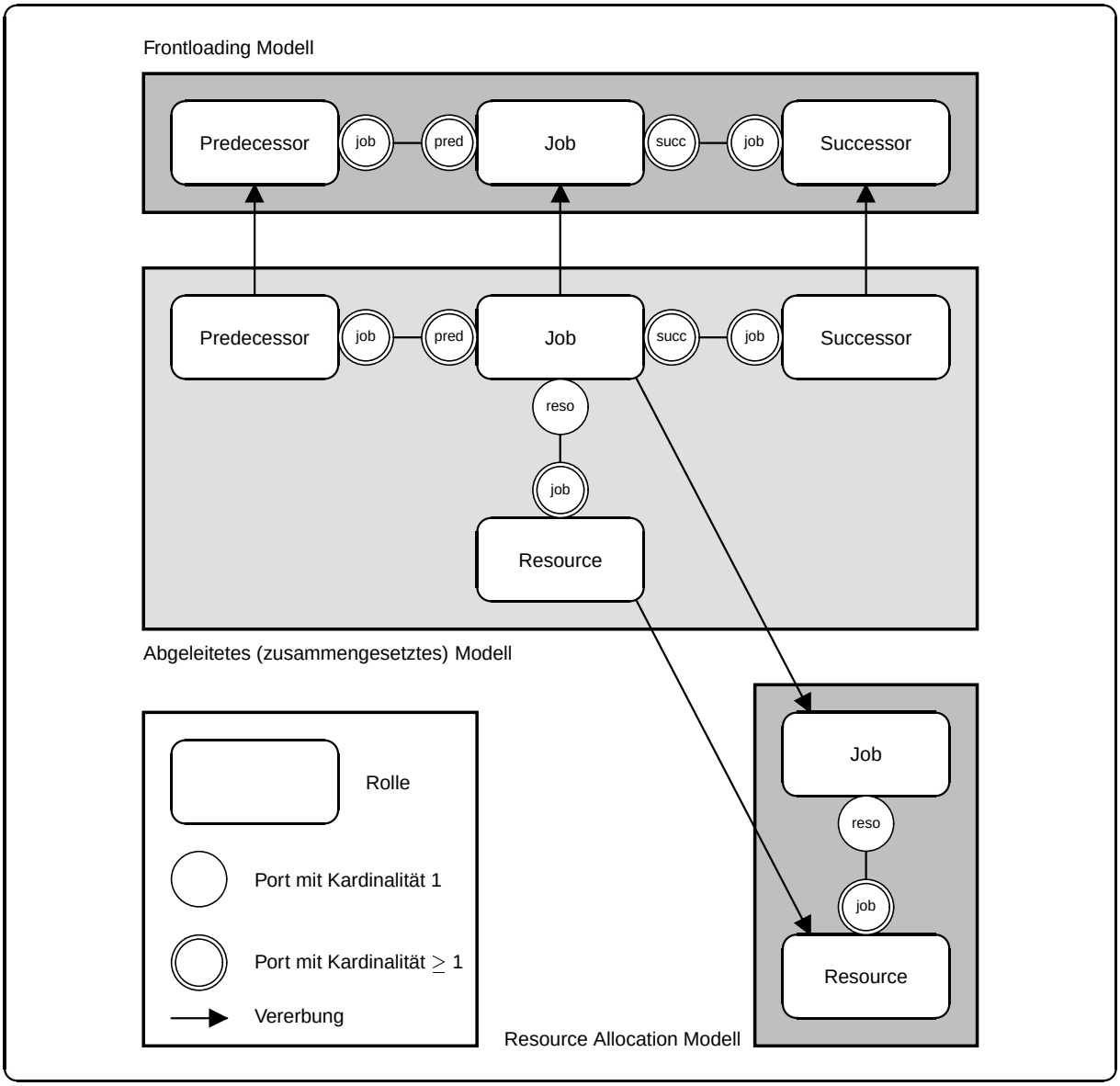


Abb. 21 Notation der Kollaborationssicht der OORAM-Methode ([RWL01, S. 24])

punkt ein bestimmter Auftrag gestartet werden kann ([RWL01, S. 19]). Ein **Port** sagt aus, daß die Rolle eine andere Rolle kennt und daher Nachrichten an diese Rolle senden kann. Beispielsweise darf die `Predecessor`-Rolle eine Nachricht an eine `Job`-Rolle senden, da die Verbindung zwischen beiden Rollen auf der Seite der `Job`-Rolle den `pred`-Port besitzt. Durch die Port-Kardinalität wird spezifiziert, ob eine Rolle mit genau einer anderen Rolleninstanz oder mit mehreren Rolleninstanzen in Beziehung steht.

Ein wesentliches Konzept innerhalb der OORAM-Methode ist die **Rollenmodellsynthese** (*Role Model Synthesis*). Durch die Verwendung des Vererbungskonzepts können mehrere Rollenmodelle in ein neues Rollenmodell integriert werden. Die Vererbung bezieht sich aber nicht nur auf einzelne Klassen, sondern auch auf die Struktur und das Verhalten der beteiligten Rollenmodelle. Das neu entstandene Rollenmodell deckt damit mehrere Zuständigkeitsbereiche ab. Abb. 21 enthält neben dem Frontloading-Modell noch das Resource-Allocation-Modell, welches ebenfalls die `Job`-Rolle besitzt. Dieses Rollenmodell spezifiziert die Zuordnung von

Betriebsmitteln zu Aufträgen. Das abgeleitete Rollenmodell ist in der Mitte von Abb. 21 (S. 43) zu sehen. Die neue `Job`-Rolle besitzt alle Attribute der `Job`-Rollen aus den beiden anderen Modellen. Die funktionale Integration findet auf der Ebene der Operationen statt. Ein Beispiel ist die Verarbeitung der Nachricht `completionTime` der `Job`-Rolle. Wenn die `Job`-Rolle alle Zeitpunkte kennt, zu denen ihre `Predecessor`-Rollen die Arbeit beendet haben, kann sie im Frontloading-Modell ihren eigenen Fertigstellungszeitpunkt berechnen und diese Informationen über eine `completionTime`-Nachricht an alle `Successor`-Rollen mitteilen. Wird das Resource-Allocation-Modell integriert, so muß die `completionTime`-Operation der `Job`-Rolle zusätzlich alle `Resource`-Rollen abfragen, wann sie die benötigten Betriebsmittel frühestens erhalten kann. Dieses Beispiel zeigt, daß eine Integration der Operationen in der Regel nicht automatisch möglich ist.

Für die Implementierung der Rollenmodelle wird vorgeschlagen, die Rollen durch Klassen umzusetzen. Hierbei ist es möglich, daß eine Klasse mehrere Rollen realisiert (z.B. könnte für die Rollen `Predecessor`, `Job` und `Successor` eine Klasse verwendet werden). Unterstützt die verwendete Programmiersprache Mehrfachvererbung, so kann diese bei der Rollenmodellsynthese genutzt werden. Beispielsweise könnten zunächst die `Job`-Rollen aus dem Frontloading-Modell und dem Resource-Allocation-Modell als einzelne Klassen implementiert werden und im nächsten Schritt die `Job`-Rolle aus dem abgeleiteten Rollenmodell entstehen, indem diese die beiden anderen `Job`-Rollen als Oberklassen verwendet.[23]

Bei der OORAM-Methode stellt der Modellierungsprozeß für große, komplexe Systeme den Schwerpunkt dar. In [And97] wird das OORAM-Rollenmodellkonzept formalisiert. Das Hauptaugenmerk liegt auf der Verhaltensmodellierung eines Rollenmodells über Zustandsautomaten. Zusätzlich wird die Semantik von Operationen auf Rollenmodellen, wie z.B. die Rollenmodellsynthese, näher untersucht.

Insgesamt läßt sich festhalten, daß die OORAM-Methode Rollenmodelle vorwiegend aus zwei Gründen einsetzt:

- Ein Rollenmodell dient zur Modellierung der Zusammenarbeit einer Objektmenge.
- Ein komplexes System kann in der Analysephase zunächst in Rollenmodelle zerlegt werden, die jeweils einen bestimmten Zuständigkeitsbereich abdecken. In einem zweiten Schritt lassen sich diese Modelle dann in übergeordnete Modelle synthetisieren.

Wesentliche Eigenschaften anderer Rollenmodelle sind im OORAM-Rollenmodell nicht enthalten:

- Rollen werden nicht dynamisch Objekten zugewiesen.
- Als direkte Konsequenz aus dem ersten Punkt gibt es auch keine Restriktionen für Rollen.
- Die Abstraktionsprozesse der Vererbung und Aggregation sind für Einzelrollen nicht definiert, sondern nur auf der Ebene der Rollenmodelle.
- Wenn die einzelnen Rollenmodelle zu einem Gesamtmodell synthetisiert wurden, geht die Eigenschaft verloren, daß über Rollen bestimmte Sichtweisen auf Objekte definiert werden. Durch die Synthetisierung werden alle Rollen eines Objekts in eine einzige Rolle integriert, die dann letztlich die Funktionalität des Gesamtobjekts repräsentiert.

[23] Bei einer konkreten Implementierung müssen natürlich die Namenskonflikte aufgelöst werden. Es kann z.B. keine drei `Job`-Klassen geben.

4.4.2 Der Ansatz von VAN HILST und NOTKIN

Bei der OORAM-Methode stand die Rollenmodellsynthese im Vordergrund. Der Ansatz von VAN HILST und NOTKIN ([VHN96b], [VHN96a]) legt den Schwerpunkt auf eine enge Verknüpfung der im Analysemodell identifizierten Rollen und deren Implementierung. Die Rollen treten explizit im Quelltext auf, um eine bessere Anpassungsfähigkeit und Wiederverwendbarkeit zu erreichen. Finden beispielsweise Anpassungen im Analysemodell statt, so soll das Auffinden der betroffenen Stellen im Quelltext durch diesen Ansatz deutlich erleichtert werden.

Auf der Analyseebene werden zunächst die für eine Applikation benötigten Objekte (und damit Klassen) identifiziert. Der zweite Schritt ermittelt Szenarios auf diesen Objekten, indem typische Anwendungsfälle (*Use Case*) für die Applikation betrachtet werden. Aus einem Szenario läßt sich dann eine Objektmenge zusammen mit den Objektbeziehungen und Methodenaufrufen ableiten. Um sich auf diejenigen Objekteigenschaften zu konzentrieren, die innerhalb eines konkreten Szenarios wichtig sind, wird der Rollenbegriff eingeführt. Als Ergebnis entsteht eine Objektkollaboration, deren Spezifikation aus einer Menge von Rollen sowie einer Beschreibung des Kontrollflusses zwischen den Rollen besteht. Die Kontrollflußspezifikation findet in [VHN96b, S. 363] über ein Interaktionsdiagramm statt, das im wesentlichen einem UML-Sequenzdiagramm entspricht. Zwischen Objekten und Rollen besteht der folgende Zusammenhang:

- Ein Objekt kann an mehreren Kollaborationen teilnehmen. Daher kann ein Objekt als eine Sammlung von Rollen aufgefaßt werden.
- Jedes Objekt, das eine für eine bestimmte Kollaboration notwendige Rolle besitzt, kann an dieser Kollaboration teilnehmen.

Für eine Abbildung der Rollen in die C^{++}-Programmiersprache ([Str00]) werden Klassen-Templates benutzt. Jeder Rolle wird ein Klassen-Template zugeordnet, das mit allen anderen Rollen aus der zugehörigen Kollaboration parametrisiert ist. Für eine Rolle `RoleA`, die mit zwei weiteren Rollen `RoleB` und `RoleC` zusammenarbeitet, hätte das Klassen-Template die folgende Grundstruktur:

```
template < class RoleBType, class RoleCType, class SuperType >
class RoleA : public SuperType {
  RoleBType *roleB;
  RoleCType *roleC;
  ...
};
```

Durch den Template-Parameter `SuperType` kann `RoleA` eine variable Oberklasse zugeordnet werden. Diese Eigenschaft unterstützt ein Zusammensetzen unterschiedlicher Rollen. Um nun die Rolle `RoleA` an eine Klasse `ClassA` zu binden, müssen zwei Bedingungen erfüllt sein:

- Für die Template-Parameter `RoleBType` und `RoleCType` existieren entsprechende Klassen-Templates, z.B. `RoleB` und `RoleC`.
- Es wurden zwei Klassen `ClassB` und `ClassC` definiert, an die die Rollen `RoleB` und `RoleC` gebunden sind.

Die Zuordnung der Rolle `RoleA` zur Klasse `ClassA` läßt sich damit wie folgt vornehmen:

```
class ClassA : public RoleA < ClassB, ClassC, EmptyClass > {};
```

Als Ergebnis liegt eine Klasse `ClassA` vor, die die Rolle `RoleA` erweitert. Ein Objekt der Klasse `ClassA` besitzt damit Referenzen zu Objekten der Klassen `ClassB` und `ClassC`, die ihrerseits die Rollen `RoleB` und `RoleC` erweitern und damit übernommen haben. Für die Rolle `RoleA` wurde als Oberklasse `EmptyClass` verwendet, die hier die leere Klasse darstellen soll, d.h. aus semantischer Sicht besitzt `RoleA` keine Oberklasse.

Da sich eine Klasse üblicherweise aus mehreren Rollen zusammensetzt, muß eine Komposition der Klasse aus ihren Rollen stattfinden. Der gewählte Ansatz basiert auf der Einfachvererbung und führt zu sehr tiefen Vererbungshierarchien[24]. Um einen Eindruck von den entstehenden Klassendefinitionen zu bekommen, wird hier die in [VHN96b, S. 364] dargestellte Struktur der Klasse `DepositReceiver` wiedergegeben[25]:

```
class DepositReceiver1Class : public
        DRAddingItemRole   < ReceiptBasisClass, DepositItemClass,
                             emptyClass > {};
class DepositReceiver2Class : public
        DRItemStuckRole    < CustomerPanelClass, AlarmClass,
                             DepositReceiver1Class > {};
class DepositReceiver3Class : public
        DRValidateItemRole < CustomerPanelClass, DepositItemClass,
                             DepositReceiver2Class> {};
typedef DepositReceiver3Class DepositReceiverClass;
```

Das von VAN HILST und NOTKIN verwendete Rollenkonzept ist bezüglich seiner Eigenschaften dem Rollenkonzept aus dem OORAM-Ansatz sehr ähnlich und besitzt damit auch dessen Einschränkungen (vgl. S. 44). Durch die vorgestellte Implementierungsstruktur entstehen sehr viele, bezogen auf die Funktionalität kleine Rollen, die in [VHN96b] daher auch als Rollenfragmente bezeichnet werden. Außerdem können sehr stark verschachtelte Template-Strukturen auftreten. Insgesamt bleibt die Frage, ob auf diesem Weg tatsächlich eine einfache und damit leicht verständliche Abbildung von Rollen aus der Analysephase in Klassendefinitionen der Implementierungsphase erreicht wurde.

4.4.3 Rollenmodellbasierter Framework-Entwurf

Bei dem Verfahren zum Framework-Entwurf von RIEHLE ([RG98], [Rie00]) stellen Klassen- und Rollenmodelle die zentralen Strukturierungseinheiten dar. Das Gesamtmodell baut auf den folgenden Definitionen auf:

- **Role Type**: Ein Rollentyp definiert das Verhalten einer Rolle, die von einem Objekt übernommen werden kann. Er definiert die Operationen und den Zustand einer Rolle ([Rie00, S. 33]).

[24] In [VHN96b, S. 367] werden 10 bis 20 Rollen (und damit Unterklassen) als typische Werte angegeben.

[25] Es handelt sich dabei um einen Neuentwurf des Automaten zur Rücknahme von leeren Getränkebehältern aus [JCJÖ92].

- **Role Model**: Ein Rollemodell besteht aus einer Menge von Rollentypen. Die zugehörigen Rollen arbeiten zusammen, um eine bestimmte Aufgabe zu erfüllen. Zusätzlich ist für ein Rollenmodell eine Menge von Rollenrestriktionen definiert ([Rie00, S. 37]).
- **Role Constraint**: Eine Rollenrestriktion besitzt einen Wert aus der Menge {`role-implied`, `role-equivalent`, `role-prohibited`, `role-dontcare`}. Jedem Paar (R, S) aus der Rollentypmenge eines Rollenmodells ist genau einer dieser Werte zugeordnet. Sei r eine Rolle des Typs R und s eine Rolle des Typs S. Dann gilt ([Rie00, S. 38]):
  - `role-implied`: Das Objekt, welches die Rolle r spielt, ist auch jederzeit in der Lage, die Rolle s zu übernehmen $\rightarrow$ die Rolle r impliziert die Rolle s.
  - `role-equivalent`: Die Rolle r impliziert s und s impliziert r.
  - `role-prohibited`: Ein Objekt, das die Rolle r spielt, darf die Rolle s nicht übernehmen.
  - `role-dontcare`: Die Rollen r und s sind unabhängig voneinander.

  Die Rollenrestriktionen wirken sich dann aus, wenn ein Objekt mehrere Rollen aus dem Rollenmodell übernehmen kann. Sie können übersichtlich in einer Rollenbeziehungsmatrix (*Role Relationship Matrix*) dargestellt werden ([Rie97]).
- **Class Model**: Ein Klassenmodell besteht aus einer Menge von Klassen. Für je zwei Klassen C_1 und C_2 gilt (zumindest) eine der beiden folgenden Beziehungen:
  - Die beiden Klassen stehen in einer Vererbungsbeziehung.
  - Die Klasse C_1 besitzt in ihrer Rollentypmenge (mindestens) einen Rollentyp, der eine Beziehung zu einem Rollentyp aus der Rollentypmenge von C_2 hat ([RG98]).

  Ein Klassenmodell besteht somit aus einer Menge von Klassen und einer Menge von Rollenmodellen, da eine Beziehung zwischen zwei Rollentypen über ein Rollenmodell definiert ist.

Die Klassenmodelle bilden jetzt die Grundlage für ein Framework. Jedes Framework besitzt genau ein Klassenmodell.[26]

Obwohl in [Rie00] eine eigenständige Kompositionsfunktion für Rollenmodelle nicht vorgesehen ist (vgl. [Rie00, S. 42]), ist bei einer Abbildung der Rollenmodelle auf die Klassen eines Klassenmodells zu beachten, daß zusätzliche Restriktionen zwischen den Rollen unterschiedlicher Rollenmodelle auftreten können (vgl. [RG98])[27]. Bei der Zuordnung von Rollentypen zu Klassen wird davon ausgegangen, daß die *Abstract Superclass Rule* ([Hür94]) gilt, die besagt, daß nur die Blattknoten einer Vererbungshierarchie konkrete Klassen sein können. Von einer konkreten Klasse darf somit keine Klasse mehr abgeleitet werden. Für die Umsetzung der Restriktionen ergibt sich damit:

- Eine `role-dontcare`-Restriktion hat keinen Einfluß darauf, ob eine Klasse bestimmte Rollentypen zur Verfügung stellt oder nicht.
- Dies gilt auch für eine `role-prohibited`-Restriktion. Die beiden betroffenen Rollenty-

[26] Da es hier um die Eigenschaften des verwendeten Rollenmodells geht, soll auf den Framework-Entwurf nicht weiter eingegangen werden. Ausführliche Informationen zu diesem Thema finden sich in [Rie00].

[27] In [RG98] wird dann doch von der expliziten Komposition von Rollenmodellen gesprochen.

pen können von der Klasse bereitgestellt werden. Es ist nur sicherzustellen, daß innerhalb einer konkreten Objektkollaboration ein Objekt dieser Klasse nicht gleichzeitig beide Rollen übernimmt.

In der Folge soll eine Restriktion zwischen den Rollentypen R und S bestehen, wobei die Klasse C_1 den Rollentyp R und die Klasse C_2 den Rollentyp S bereitstellt.

- Für eine `role-implied`-Restriktion muß eine der folgenden Bedingungen erfüllt sein:
  - $C_1 = C_2$
  - C_2 ist eine Superklasse von C_1.
  - Jede konkrete Unterklasse von C_1 stellt den Rollentyp S zur Verfügung.

  Damit ist sichergestellt, daß ein Objekt, welches eine Rolle des Typs R spielt, auch eine Rolle des Typs S übernehmen kann.
- Für eine `role-equivalent`-Restriktion muß eine der folgenden Bedingungen erfüllt sein:
  - $C_1 = C_2$
  - C_2 ist eine Superklasse von C_1 und jede konkrete Unterklasse von C_2 stellt den Rollentyp R zur Verfügung.
  - C_1 ist eine Superklasse von C_2 und jede konkrete Unterklasse von C_1 stellt den Rollentyp S zur Verfügung.

  Damit ist garantiert, daß ein Objekt, welches eine Rolle mit einem der Rollentypen R oder S annimmt, immer auch eine Rolle des anderen Rollentyps spielen kann.

Da eine Klasse mehrere Rollentypen bereitstellen kann, ist der zugehörige Klassentyp eine Komposition der Rolltentypen aus der Rollentypmenge dieser Klasse. Für die Spezifikation des Klassentyps werden zwei prinzipielle Varianten vorgeschlagen:

- Es kann ein Typspezifikationsmechanismus verwendet werden, der für eine Rollenmodellierung und -integration geeignet ist (z.B. [LW93] und [LW94]).
- Der Anwendungsentwickler führt die Komposition bei der Implementierung der Klassentypen durch, indem er die Eigenschaften der einzelnen Rollentypen in den Klassentyp integriert.

Um die zweite Variante umzusetzen, werden für die Sprachen JAVA, C^{++} und SMALLTALK die folgenden Ansätze vorgeschlagen:

- JAVA: Für ein Klassenmodell wird ein Paket definiert, das damit einen Namensraum für dieses Klassenmodell zur Verfügung stellt. Jedem Rollentyp wird ein Interface zugeordnet. Eine Klasse muß dann alle Interfaces implementieren, deren Rollentypen zur Rollentypmenge der Klasse gehören ([Rie00, S. 91]).
- C^{++}: Das JAVA Interface Konzept wird durch virtuelle Funktionen und Mehrfachvererbung emuliert ([Rie00, S. 94]).
- SMALLTALK: Die Mehrfachvererbung und Namensräume werden simuliert, um die JAVA-Implementierungsstruktur umsetzen zu können ([Rie00, S. 95]).

Da Frameworks sehr langlebige Software-Einheiten darstellen, ist es beim Entwurf mit großer Wahrscheinlichkeit nicht möglich, alle Anwendungskontexte vorauszusehen. Um nicht nur statisch (d.h. zum Übersetzungszeitpunkt), sondern auch dynamisch neue Rollentypen in das Framework integrieren zu können, wird das *Role-Object*-Muster vorgeschlagen ([BRSW97], [BGK^{+}97]), dessen Struktur und Eigenschaften in Abschnitt 4.6.8 (S. 66) besprochen werden.

Die Eigenschaften des verwendeten Rollenmodells orientieren sich wie bei der OORAM-Methode und dem Ansatz von VAN HILST und NOTKIN wieder sehr stark an der Modellierung von Objektkollaborationen. Die wesentlichen Unterschiede zu den beiden vorgenannten Ansätzen sind:

- Durch die Verwendung des Role-Object-Musters ist eine dynamische Integration von Rollen zur Laufzeit möglich.
- Für Rollen lassen sich Restriktionen formulieren. Allerdings hat nur die `role-prohibited`-Restriktion eine Auswirkung auf das Laufzeitverhalten von Objekten. Die beiden anderen Restriktionen[28] sind der Entwurfsebene zuzuordnen.

Ein sehr ähnlicher Ansatz wird von ZHAO und KENDALL in [ZK00] und [ZK01] vorgestellt. Dort wird die Rollenmodellierung für den Komponentenentwurf eingesetzt. Eine Komponente wird hierbei im Sinne von PFISTER und SZYPERSKI aufgefaßt: *Eine Komponente ist definiert als eine Sammlung von kooperierenden Objekten, mit einer klar definierten Grenze zu anderen Objekten oder Komponenten* ([PS96]). Die Rollenmodelle dienen auch hier wieder zur Spezifikation der Objektinteraktionen innerhalb einer Komponente.

4.4.4 Das Epsilon Modell

TAMAI stellt in [Tam02] und [Tam99] einen Ansatz vor, in dem Objekte und Rollen als gleichberechtigte Konzepte betrachtet werden. Die Grundidee besteht in der Definition von **Kollaborationsbereichen** (*Collaboration Field*). Ein Kollaborationsbereich beschreibt eine Umgebung, in der verschiedene, zusammenarbeitende Rollen existieren. Objekte können sich dynamisch an Rollen einzelner oder mehrerer Kollaborationsbereiche binden. Kollaborationsbereiche sollten derart entworfen werden, daß sie einen bestimmten Zuständigkeitsbereich abdecken. Die Rollen werden nach außen durch ihren Kollaborationsbereich gekapselt. Sie können untereinander kommunizieren, haben aber nicht die Möglichkeit, auf Rollen anderer Kollaborationsbereiche zuzugreifen. Aufgrund dieser Eigenschaft sind Kollaborationsbereiche geeignete Einheiten für eine Wiederverwendung. Zur Umsetzung des Konzepts wird an der Programmiersprache EPSILONJ gearbeitet, die eine JAVA-ähnliche Syntax besitzt. Ein Kollaborationsbereich wird mit dem Schlüsselwort `context` eingeleitet. Das folgende Beispiel definiert einen Kollaborationsbereich `Company`, der die beiden Rollen Arbeitgeber (`Employer`) und Arbeitnehmer (`Employee`) enthält.

```
context Company {
  role Employer {
    int salary = 100;
    void pay() {
      Employee.getPaid(salary);
    }
  }
  role Employee {
    int deposit;
```

[28] Die `role-dontcare`-Restriktion stellt im eigentlichen Sinne keine Restriktion dar, weil sie keine Einschränkung für die beteiligten Rollen definiert.

```
    void getPaid (int salary) {
      deposit += salary;
    }
  }
}
```

Ein `context` läßt sich instanziieren. Dabei wird von jeder Rolle automatisch eine Instanz erzeugt, die über ihren Rollennamen referenzierbar ist. Es ist auch möglich, mehrere Instanzen eines Rollentyps zu erzeugen[29]. Ein Objekt kann dynamisch an eine Rolle gebunden werden, wodurch das Objekt jetzt auch die Attribute und Methoden der Rolle besitzt. Zusätzlich ist es möglich, Attribute und Methoden zu ersetzen oder umzubenennen. In der Fortsetzung des obigen Beispiels wird eine Klasse `Person` definiert. Bei dem Binden eines Personenobjekts an die `Employee`-Rolle erfolgt die Ersetzung des Attributs `deposit` aus der Rolle `Employee` durch das Attribut `money` aus der Klasse `Person`.

```
class Person {
  int money;
}

Person Tanaka = new Person();
Company tadai = new Company();
tadai.Employee.bind(Tanaka)
  replacing Employee.deposit
       with Tanaka.money;
```

Im Unterschied zu den bereits vorgestellten Kollaborationsmodellen findet sich das verwendete Rollenkonzept auch auf der Programmiersprachenebene wieder. Die prinzipiellen Eigenschaften des Rollenkonzepts sind aber mit denjenigen der anderen Kollaborationsmodelle vergleichbar.

[29] Es wird allerdings nicht erklärt, wie die einzelnen Instanzen dann unterschieden werden können.

4.5 Rollenkonzepte im Umfeld von Agentensystemen

4.5.1 Eigenschaften von Software-Agenten

Software-Agenten[30] stellen ein vergleichsweise neues Programmierparadigma zur Realisierung verteilter Applikationen dar. In [Ode00, S. 16] wird ein Agent als *eine autonome Software-Einheit* bezeichnet, *die mit ihrer Umgebung interagieren kann*. Wichtige Eigenschaften von Agenten sind ([Obj00, S. 8]):

- **Autonomie** (*autonomous*): Ein Agent kann ohne direkte äußere Einflußnahme agieren.
- **Zielorientierung** (*proactive*, *goal-oriented*): Ein Agent reagiert nicht nur auf seine Umgebung, sondern verfolgt ein Ziel.
- **Intelligenz** (*intelligent*): Der interne Zustand ist (zumindest teilweise) durch Wissen formalisiert (z.B. Ziele, Pläne und Annahmen) und die Interaktion mit anderen Agenten erfolgt unter Verwendung einer symbolischen Sprache.
- **Adaptivität** (*adaptive*): Der Agent ist in der Lage, sein Verhalten auf der Grundlage von Erfahrungen anzupassen.
- **Interaktion** (*interactive*): Der Agent kommuniziert mit seiner Umgebung und anderen Agenten.
- **Kooperation** (*cooperative*): Ein Agent arbeitet mit anderen Agenten zusammen, um ein gemeinsames Ziel zu erreichen.[31]
- **Koordination** (*coordinative*): Agenten werden oft über (z.B.) Pläne oder Workflows koordiniert.
- **Mobilität** (*mobile*): Der Agent besitzt die Fähigkeit, sich selbst von seiner aktuellen Umgebung in eine neue Umgebung zu transportieren.

Es existiert noch keine einheitliche Sicht bezüglich der Frage, welche Eigenschaften aus der obigen Liste ein Agent erfüllen muß. Allerdings enthalten nahezu alle Definitionen die Eigenschaften Autonomie, Interaktion und Adaptivität ([Obj00, S. 9]). Arbeiten mehrere Agenten zur Erreichung eines gemeinsamen Ziels zusammen, liegt ein Multi-Agentensystem vor.

Aus den Anforderungen Interaktion, Kooperation und Koordination läßt sich erkennen, daß bei Agentensystemen die Modellierung von Kommunikationsbeziehungen einen zentralen Aspekt darstellt. Daher bietet es sich an, hierfür ein Rollenkonzept zu verwenden. Da Rollen dynamisch von Objekten übernommen und wieder aufgegeben werden können, ist ein Rollenkonzept auch sehr gut für die Realisierung der Mobilitäts- und Adaptivitätseigenschaften von Agenten geeignet ([Ken01]).

4.5.2 Der Ansatz von KENDALL

In [Ken00b], [Ken01] und [Ken99a] werden Rollenmodelle als Abstraktion für die Analyse, den Entwurf und die Implementierung von Agentensystemen vorgeschlagen, wobei der Schwerpunkt auf nicht mobilen Agenten liegt. Für die Durchführung der Analysephase werden zwei

[30] In der Folge wird stattdessen der Begriff Agent verwendet.
[31] Statt Kooperation wird auch der Begriff **Kollaboration** (*collaborative*) verwendet.

alternative Vorgehensweisen vorgeschlagen:

- Identifikation der relevanten Agenten einer Applikation und ihrer Interaktionen: Diese sind dann als Rollenmodelle zu formulieren, wobei jedes Rollenmodell einen bestimmten Systemaspekt abdeckt.
- Verwendung eines Rollenmodellkataloges: Dieser ähnelt in seiner Struktur einem Musterkatalog, der häufig verwendete Entwurfsmuster beschreibt. Die Applikation wird systematisch nach in dem Katalog dokumentierten Rollenmodellen durchsucht.

In einem zweiten Schritt ist zu analysieren, welche Abhängigkeiten zwischen den identifizierten Rollenmodellen bestehen. Abhängigkeiten kommen darüber zum Ausdruck, daß später einem Objekt Rollen aus mehreren Rollenmodellen zugeordnet werden können oder müssen.

Die Entwurfsphase identifiziert diejenigen Rollen, die ein gegebener Agent übernehmen muß. Für diese Rollen ist eine Komposition vorzunehmen.

Als Implementierungskonzepte werden das *Role-Object*-Muster ([BRSW97]) oder die Anwendung der *aspektorientierten Programmierung* ([KIL+97]) vorgeschlagen. Die Eigenschaften des Role-Object-Ansatzes werden in Abschnitt 4.6.8 (S. 66) untersucht, diejenigen der Aspektorientierung in Abschnitt 4.7.3 (S. 75).

Um mobile Agenten zu unterstützen, wird die Idee einer *Plug-in*-Architektur vorgestellt (siehe [Ken00b, S. 37]). Da die Gesamtgröße (d.h. der Speicherplatzbedarf) eines mobilen Agenten ein wichtiges Kriterium sein kann, sollte der Agent nur diejenigen Rollen besitzen, die er in seiner neuen Umgebung benötigt. Ein ähnlicher Ansatz wird auch in [TGML98] verfolgt. Auch hier kann rollenspezifische Funktionalität dynamisch zur Laufzeit des Agenten nachgeladen werden.

4.5.3 Der XROLE-Ansatz

CABRI, LEONARDI und ZAMBONELLI gehen in ihrem Ansatz von einem Multi-Agentensystem aus ([CLZ02b]). Agenten kommunizieren zur Erreichung eines gemeinsamen Ziels sowohl mit anderen Agenten als auch mit ihrer Umgebung. Das verwendete Rollenkonzept besitzt die folgenden Eigenschaften ([CLZ02a]):

- Rollen sind unabhängig von Agenten und Applikationen definiert. Sie werden als generisch (*generic*) bezeichnet, weil sie bestimmte Eigenschaften zur Verfügung stellen, die in unterschiedlichen Applikationen von Agenten nutzbar sind.
- Eine Rolle hat einen temporären (*temporary*) Charakter, da sie von einem Agenten nur für einen bestimmten Zeitraum angenommen wird.
- Eine Rolle definiert sich über eine Menge von Fähigkeiten (*Capability*) und ein (nach außen) sichtbares Verhalten (*Expected Behavior*).
  - Die Fähigkeiten werden durch Aktionen (*Action*) realisiert. Ein Agent, der eine Rolle übernimmt, kann diese Aktionen ausführen.
  - Die Modellierung des erwarteten Verhaltens findet ereignisbasiert statt. Für jede Rolle ist eine Menge von Ereignissen (*Event*) definiert, auf die die Rolle reagiert.

Zur Spezifikation von Rollen innerhalb des XROLE-Systems wird XML[32] benutzt. Alle Rollenbeschreibungen müssen einer XML-Schemadefinition genügen. Damit ist die syntaktische Korrektheit sichergestellt. Zur Unterstützung der Implementierungsphase lassen sich die XML-Rollendefinitionen unter Verwendung eines XSLT-Dokuments[33] beispielsweise in JAVA Interfaces übersetzen. Die in [Cab01] vorgeschlagene Infrastruktur für agentenbasierte Applikationen besteht aus vier Ebenen:

- **Agentenebene** (*Agent Level*): Agenten existieren in einem Netzwerk, das aus einer Menge von Knoten besteht, wobei jeder Knoten eine eigenständige Umgebung für die Agenten definiert.
- **Infrastrukturebene** (*Infrastructure Level*): Hier wird die Umgebung für Agenten beschrieben. Eine Umgebung enthält eine Menge von Rollen, die Agenten zur Verfügung stehen, die diese Umgebung betreten.
- **Strategie- und Mechanismusebene** (*Policy and Mechanism Level*): In dieser Ebene werden die Mechanismen für die Interaktion zwischen den Agenten definiert, d.h. insbesondere der verwendete Kommunikationsmechanismus. Eine Strategie könnte beispielsweise festlegen, ob zwei Rollen direkt miteinander kommunizieren dürfen oder nicht.
- **Betriebsmittelebene** (*Resource Level*): Diese Ebene beschreibt Betriebsmittel, die auf einem Knoten vorgegeben sind (z.B. das installierte Betriebssystem). Es ist die Aufgabe der Strategie- und Mechanismusebene, die Betriebsmittel den Rollen in einer geeigneten Form zur Verfügung zu stellen.

4.5.4 ROLEEP: Role Based Evolutionary Programming

Mit dem ROLEEP-Konzept ([UT01]) werden die Ideen aus dem EPSILON-Modell (vgl. Abschnitt 4.4.4, S. 49) auf kooperative, verteilte mobile Anwendungen erweitert. Das Rollenkonzept übernimmt hierbei zwei Aufgaben: zum einem ist die Mobilitätsfunktionalität an Rollen geknüpft, zum anderen dienen Rollen zur Realisierung der Kollaboration von Objekten. ROLEEP verwendet vier Grundabstraktionen, deren Zusammenspiel mit der folgenden Grammatik beschrieben wird:

```
environment ::= [ environment attributes, environment methodes,
                  roles ]
role        ::= [ role attributes, role methods,
                  binding-interface ]
agent       ::= [ roles, object ]
object      ::= [ attributes, methods ]
```

- **Umgebung** (*Environment*): Eine Umgebung stellt über ihre Attribute und Methoden den Rollen und Objekten allgemeine Dienstleistungen zur Verfügung. Ein Beispiel ist der *Role-Lookup*-Dienst. Eine Umgebung sollte inhaltlich einen abgegrenzten Zuständigkeitsbereich der Gesamtapplikation abdecken. Eine Instanz einer Umgebung kann sich über mehrere

[32] XML: Extensible Markup Language ([WWW00])

[33] XSL: The Extensible Stylesheet Language Family
XSL ist eine Familie von Empfehlungen, um die Transformation und Präsentation von XML-Dokumenten vorzunehmen ([WWW03]). XSL setzt sich aus drei Teilen zusammen: XSL TRANSFORMATION (XSLT, [WWW99]), XML PATH LANGUAGE (XPATH) und XSL FORMATTING OBJECTS (XSL-FO).

Rechnerknoten (*Host*) erstrecken.[34] Ein Beispiel ist die *Roaming*-Umgebung. Eine Instanz dieser Umgebung legt fest, innerhalb welcher Host-Menge sich Agenten bewegen können. Außerdem stellt sie die notwendigen Rollen zur Verfügung, um die Mobilitätsfunktionalität zu realisieren.

- **Rolle** (*Role*): Eine Rolle besitzt grundsätzlich die Fähigkeit, sich zwischen den Rechnerknoten einer Umgebung zu bewegen. Sie besteht aus Attributen, Methoden und einem *Binding-Interface*. Dieses wird verwendet, wenn sich ein Objekt an eine Rolle binden will.
- **Objekt** (*Object*): Ein Objekt besteht ebenfalls aus Attributen und Methoden. Ein Objekt allein besitzt keine Mobilitätsfunktionalität.
- **Agent** (*Agent*): Ein Objekt wird zu einem Agenten, indem es sich innerhalb einer Umgebung an eine Rolle bindet. Ein Objekt kann gleichzeitig an Rollen mehrerer Umgebungen gebunden sein, da sich die Rechnerknotenmengen der Umgebungsinstanzen überschneiden dürfen. Agenten referenzieren sich untereinander über die Identifikationen der aktuell zugeordneten Rolleninstanzen. Allerdings können nur Rolleninstanzen miteinander kommunizieren, die zu derselben Umgebungsinstanz gehören.
- **Binding-Interface**: Das Binding-Interface einer Rolle beschreibt die Schnittstelle, d.h. die Methoden, die diese Rolle den anderen Rollen der zugehörigen Umgebung zur Verfügung stellt. Ein Objekt, das sich an eine Rolle binden will, muß für jede Methode der Schnittstelle eine korrespondierende Methode anbieten. Da die korrespondierenden Methoden unterschiedliche Namen besitzen dürfen, besteht der Bindevorgang in einer Namensabbildung.

Für die Umsetzung des RoleEP-Konzepts findet das Epsilon/J-Framework Anwendung. Es werden drei Klassen `Environment`, `Role` und `EpsilonObj` bereitgestellt, die die Grundfunktionalität für Umgebungen, Rollen und Objekte definieren. Die `Role`-Klasse erweitert die Klasse `Aglet` aus dem Java-basierten *Aglet* Framework ([Ven97]). Eine Java-Klasse, die eine Unterklasse von `Aglet` ist, besitzt die Fähigkeit, im Netz zu migrieren. Im Unterschied zu `Applet`-Klassen kann eine Java-Aglet-Instanz aber bei der Migration ihren Zustand *mitnehmen*. Dieses ist die Voraussetzung, um das Konzept der mobilen Agenten umsetzen zu können.

4.5.5 Bewertung der Rollenkonzepte aus dem Agentenumfeld

Alle verwendeten Rollenmodelle besitzen ihren Schwerpunkt in der Modellierung von Objektkollaborationen und damit auch in der Trennung von Zuständigkeiten. Weitere Eigenschaften sind:

- Rollen können dynamisch von Objekten angenommen und wieder abgelegt werden.
- Rollen dienen einer optimierten Resourcen-Nutzung, wobei hier die Dynamik von Rollen wieder eine wesentliche Grundlage bildet.
  - Sowohl beim Kendall- als auch beim XRole-Ansatz werden auf den einzelnen Rechnerknoten nur genau diejenigen Rollen zur Verfügung gestellt, die zur Realisierung der auf diesem Knoten gewünschten Funktionalität erforderlich sind. Zwischen den Knoten werden Agenten migriert, die potentiell alle Rollen übernehmen könnten.

[34] Wie die Zuordnung zwischen einer Umgebungsinstanz und der Host-Menge erfolgt, wird in [UT01] allerdings nicht näher beschrieben.

  - Die Alternative wäre, daß ein Agent alle Rollen, die er prinzipiell spielen kann, immer mitnimmt, was aber einen erhöhten Bandbreitenbedarf bei der Übertragung zur Folge hätte. Zusätzlich würde auf dem Zielknoten mehr Speicherplatz belegt.
- Rollen können zur Realisierung der Mobilitätsfunktionalität verwendet werden, wie dies im ROLEEP-Ansatz erfolgt.
- Rolleninstanzen können in einer Kardinalität größer als 1 vorliegen. Der ROLEEP-Ansatz ist hierfür ein Beispiel.
- Restriktionen werden explizit nur im XROLE-Ansatz angeboten. Ihre Definition ist in der Strategie- und Mechanismusebene enthalten.

Keines der vorgestellten Rollenmodelle realisiert alle der oben aufgezählten Eigenschaften. Außerdem fehlt allen die Eigenschaft allgemeiner Rollenmodelle, ein Vererbungskonzept zwischen Rollen zu unterstützen. Damit ist es nicht möglich, daß eine Rolle Unterrollen annehmen kann. Umgekehrt ist es dann auch nicht möglich, daß eine Rolle eine oder mehrere Oberrollen besitzt.

4.6 Anwendung von Analyse- und Entwurfsmustern

4.6.1 Einleitung

In dem Artikel *Dealing with Roles* von MARTIN FOWLER ([Fow97b]) werden mehrere Ansätze zu einer prinzipiellen Umsetzung des Rollenkonzepts beschrieben und bewertet. Obwohl in der Artikeleinleitung davon gesprochen wird, daß es sich um einen Artikel über Analysemuster handelt, werden konkrete Vorschläge zur Implementierung des Rollenkonzepts vorgestellt.

4.6.2 Der Single-Role-Type-Ansatz

Diese Variante kann als der einfachste Implementierungsansatz aufgefaßt werden und läßt sich unter den folgenden Randbedingungen anwenden:

- Es liegt lediglich eine zweistufige Rollenhierarchie vor.
- Die Unterrollen der Oberrolle besitzen selbst keine weiteren Attribute und Operationen.
- Zwischen den Unterrollen besteht eine **exor**-Restriktion.

Ein Anwendungsbeispiel ist die Differenzierung von Personen nach ihrem Beruf. In Abb. 22 ist die Implementierungsstruktur dargestellt.

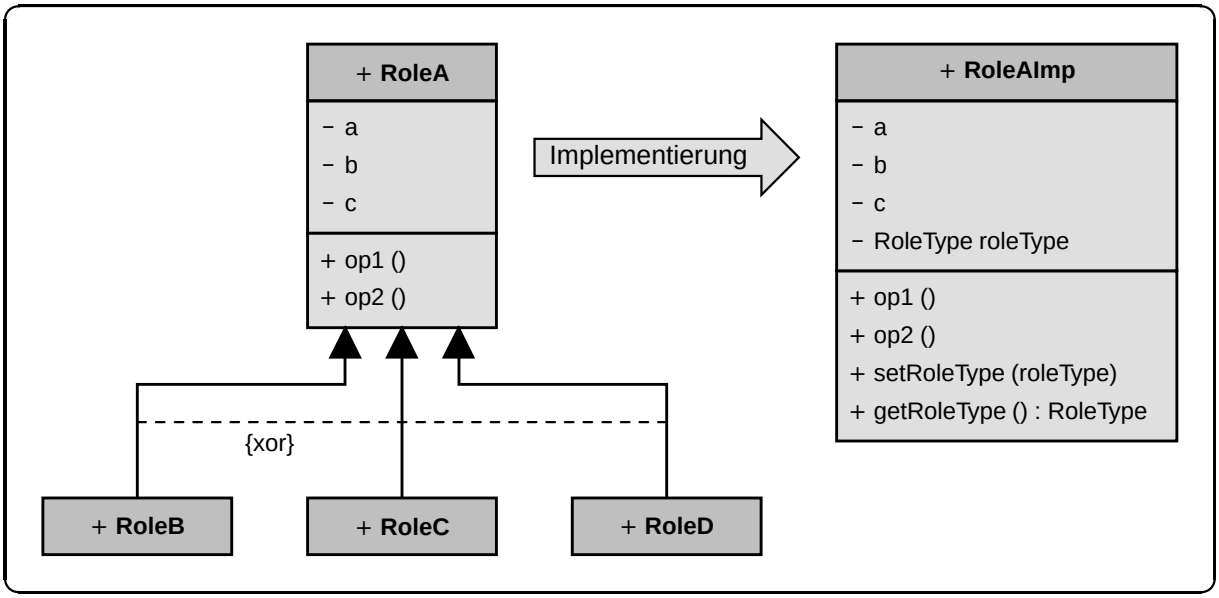


Abb. 22 Der Single-Role-Type-Ansatz

Das gesamte Rollenmodell wird durch eine einzige Klasse (**RoleAImp**) realisiert, die die Attribute und Methoden aus der Wurzelrolle (**RoleA**) übernimmt. Als weiteres Attribut kommt `roleType` hinzu, das anzeigt, welche Rolle das Objekt gerade besitzt. Der Typ `RoleType` sollte als Aufzählungstyp definiert werden, der sinnvollerweise als Werte die Namen der Unterrollen von **RoleA** besitzt. Als weitere Operationen verwendet **RoleAImp** `setRoleType` und `getRoleType`. Über `setRoleType` kann das Objekt die Rolle wechseln.

4.6.3 Der Separate-Role-Type-Ansatz

Die zweite Möglichkeit zur Umsetzung eines Rollenmodells besteht in der Implementierung separater Klassen für die einzelnen Rollen. Für das Rollenmodell aus Abb. 23 (S. 58) entstehen sechs Rollenklassen **RoleAImp** bis **RoleABCEImp**. Auf diesem Weg lassen sich Rollenmodelle mit beliebig vielen Hierarchieebenen realisieren. Auch eine Mehrfachvererbung bei Rollen kann umgesetzt werden: die Rolle **RoleE** entsteht durch die Erweiterung der Rollen **RoleB** und **RoleC**, die ihrerseits wieder Rollen von **RoleA** sind. Damit besitzt **RoleE** alle Eigenschaften und Operationen von **RoleB**, **RoleC** und **RoleA**. Wie die Redefinition und Mehrfachdefinition von Operationen behandelt wird, liegt vollständig im Ermessen des Anwendungsentwicklers. Wenn beispielsweise die Rollenklassen **RoleB** und **RoleC** jeweils eine Operation mit derselben Signatur besitzen, dann muß bei der Implementierung entschieden werden, ob die Operation aus **RoleB** oder **RoleC** übernommen wird. Alternativ könnte man Mehrfachdefinitionen von Operationen natürlich auch verbieten. Der entscheidende Nachteil dieses Ansatzes liegt aber in der Datenredundanz und der damit verbundenen Konsistenzproblematik, wenn mehrere Rollen gleichzeitig angenommen werden. Übernimmt ein Objekt der realen Welt gleichzeitig die Rollen **RoleB** und **RoleC**, so werden ihm in der Implementierung zwei Objekte der Klassen **RoleABImp** und **RoleACImp** zugeordnet. Damit sind für alle Attribute von **RoleA** jeweils zwei Werte abgespeichert. Ein weiterer Nachteil ist, daß es keinen direkten Bezug mehr zwischen den Unterollen und der Wurzelrolle gibt. Ob zwei Objekte der Rollenklassen **RoleABImp** und **RoleACImp** demselben **RoleA**-Objekt des Rollenmodells zugeordnet sind, kann nur indirekt aus der Gleichheit der **RoleA**-Attribute geschlossen werden. Besitzt keines der Attribute von **RoleA** die Schlüsseleigenschaft[35], dann läßt sich aus der Gleichheit der Attributwerte nicht mehr ableiten, daß es sich auch wirklich um dasselbe **RoleA**-Objekt des Rollenmodells handelt.

4.6.4 Der Subtype-Ansatz

Bei dieser Variante erfolgt die Implementierung der Rollenklassen unter Verwendung des Vererbungskonzepts. Für ein Objekt, welches z.B. die Rollen **RoleA**, **RoleD** und **RoleE** besitzt, wird die Klasse **RoleADEImp** bereitgestellt (vgl. Abb. 24 (S. 60)). Dieser Ansatz besitzt zwei gravierende Nachteile:

- Schon bei Rollenmodellen mit wenigen Rollen müssen sehr viele Klassen für die Implementierung bereitgestellt werden, da für jede erlaubte Kombination von Rollen eine eigene Klasse benötigt wird. In dem Beispiel aus Abb. 24 (S. 60) entstehen aus dem Rollenmodell mit 5 Rollen bereits 12 Klassen. Hat eine Rolle n direkte Unterrollen, die unabhängig voneinander angenommen werden können, weil zwischen ihnen keine **exor**-Restriktionen existieren, dann sind $2^n - 1$ Klassen zu implementieren, d.h. für $n = 10$ sind dies 1023 Klassen.
  Stellt die Implementierungssprache nur Einfachvererbung zur Verfügung, tritt zusätzlich das Problem auf, daß sehr viele Operationen mehrfach implementiert werden müssen. In dem Beispiel aus Abb. 24 (S. 60) wird die Operation `mE` 6 mal benötigt. Dieses Problem kann

[35] Die Schlüsseleigenschaft für ein Attribut besagt, daß zwei beliebige Objekte dieses Typs bei diesem Attribut unterschiedliche Werte besitzen müssen.

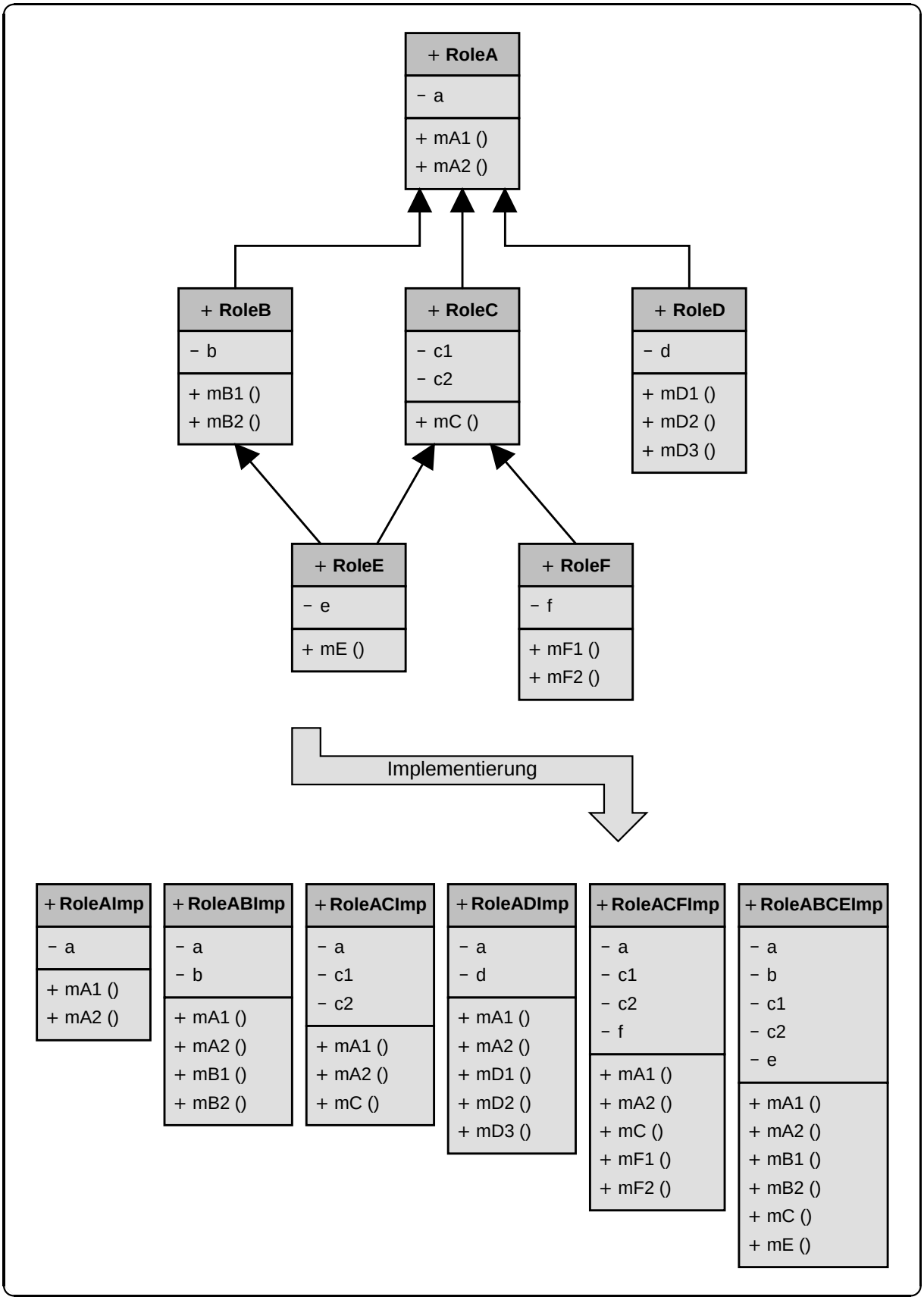


Abb. 23 Der Separate-Role-Type-Ansatz

durch die Verwendung der Mehrfachvererbung gelöst werden. In Abb. 25 (S. 61) ist die Implementierung der Klassen **RoleABImp** und **RoleABDEImp** unter Verwendung der Mehrfachvererbung dargestellt.

- Der zweite Nachteil betrifft die Tatsache, daß in gängigen objektorientierten Programmiersprachen der Typ eines Objekts bei seiner Erzeugung festgelegt wird. Damit ist es nur möglich, zur Laufzeit eine neue Rolle zu übernehmen, indem auch ein neues Objekt erzeugt wird. Besitzt ein Objekt z.B. die Rollen **RoleA** und **RoleB**, so liegt auf der Implementierungsebene ein Objekt der Klasse **RoleABImp** vor. Um jetzt zusätzlich die Rolle **RoleE** zu übernehmen, wären die folgenden Schritte notwendig: (1) Ein neues Objekt der Klasse **RoleABEImp** wird erzeugt, (2) der Zustand aus dem **RoleABImp**-Objekt muß übernommen werden, (3) alle Referenzen auf das **RoleABImp**-Objekt sind durch Referenzen auf das neue **RoleABEImp**-Objekt zu ersetzen und (4) das **RoleABImp**-Objekt ist zu löschen, sofern dies nicht automatisch durch den Garbage Collector der verwendeten Implementierungssprache erfolgt. Insbesondere der dritte Schritt ist ausgesprochen problematisch, da in Sprachen wie JAVA oder C^{++} keine Operationen zur Verfügung stehen, die für ein Objekt den Zugriff auf alle existierenden Referenzen erlauben.

In den nächsten drei Abschnitten werden drei Ansätze vorgestellt, die als eine Simulation des Subtype-Ansatzes aufgefaßt werden können und versuchen, dessen Nachteile zu vermeiden.

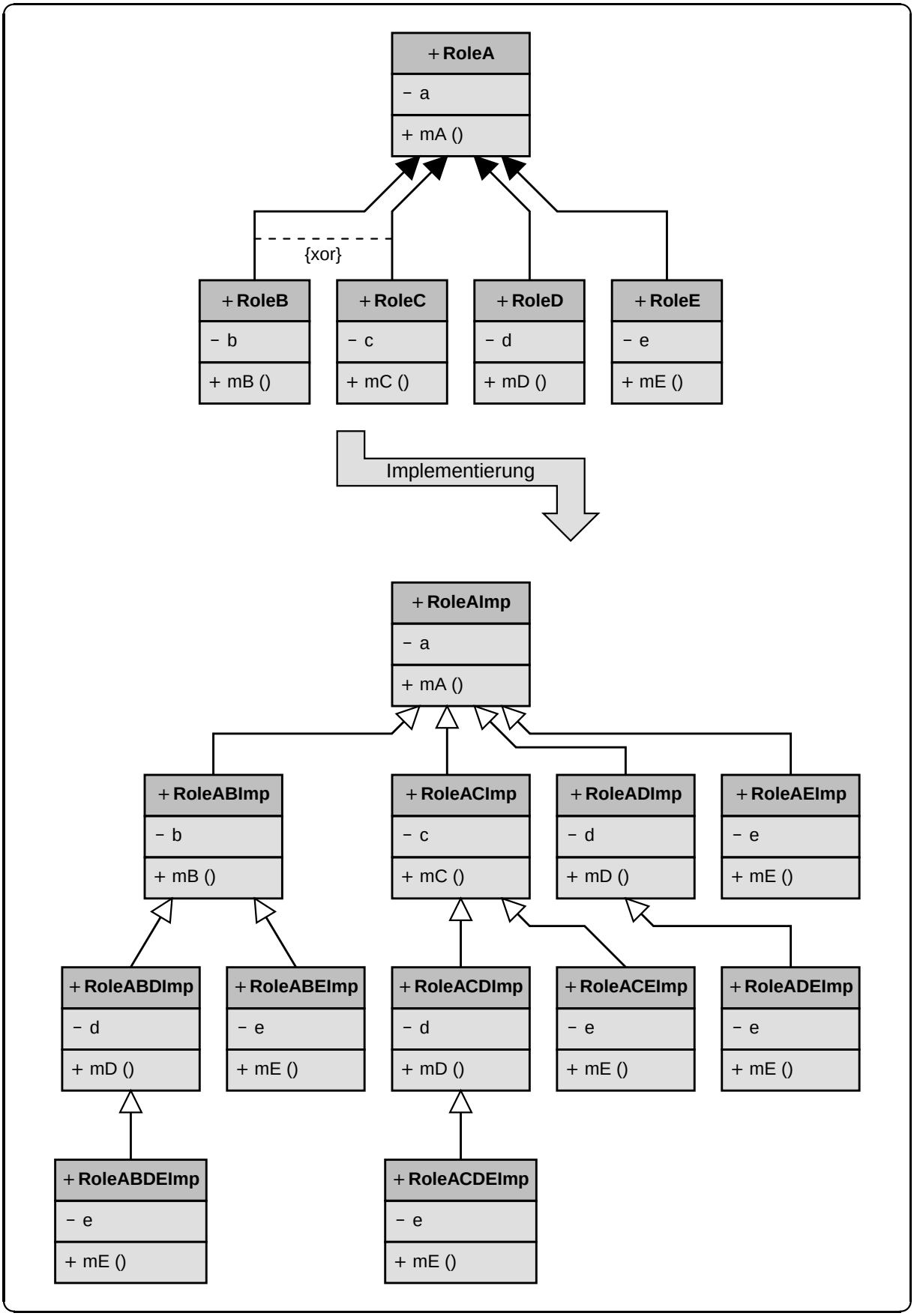


Abb. 24 Der Subtype-Ansatz

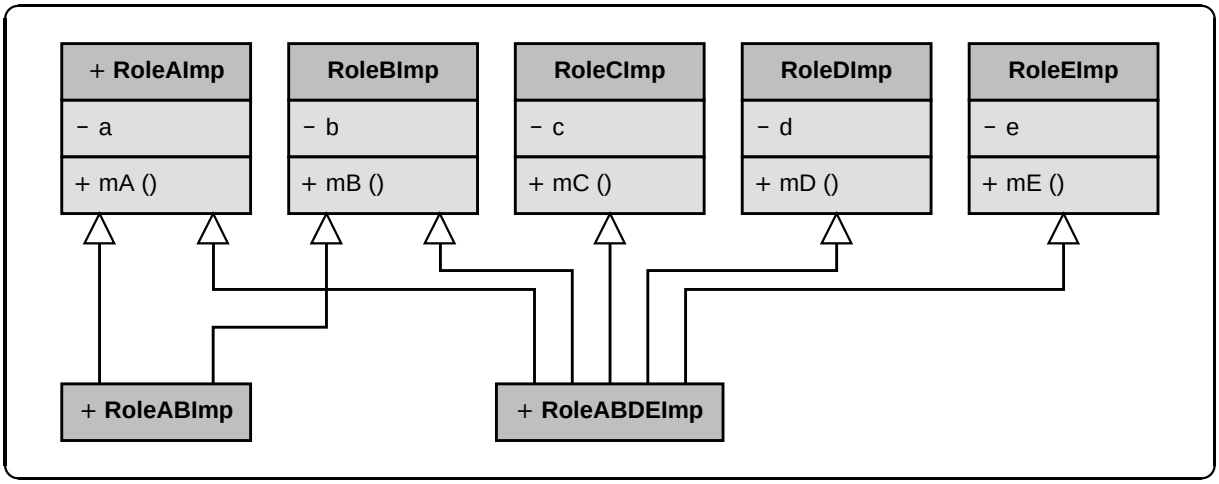


Abb. 25 Der Subtype-Ansatz unter Verwendung der Mehrfachvererbung

4.6.5 Der Internal-Flag-Ansatz

Das Rollenmodell wird durch eine einzige Klasse implementiert (vgl. auch [Fow97a, S. 283f]). Diese Klasse bietet nach außen alle Operationen an, die in den verschiedenen Rollen definiert sind (vgl. Abb. 26 (S. 62)). Für jede Menge von Rollen, die einer **exor**-Restriktion unterliegen, wird ein Flag-Attribut definiert. Im Beispiel aus Abb. 26 (S. 62) sind dies `flagBC` für die Rollen **RoleB** und **RoleC** sowie `flagD` für die Rolle **RoleD**. Hat ein Flag den Wert `true`, so liegt eine entsprechende Rolle vor. Zur Erzeugung einer Rolle des Typs `X` wird die Operation `makeX` bereitgestellt, die die Parameter des Konstruktors von **RoleX** übernimmt. Durch den Aufruf von `makeX` auf einem **RoleAImp**-Objekt wird die neue Rolle des Typs `X` erzeugt, wobei in `makeX` durch die Kontrolle des zugehörigen Flags zu überprüfen ist, ob die Objekterzeugung überhaupt erlaubt ist. Die Operation `isX` liefert die Information, ob bereits eine `X`-Rolle existiert. Da ein **RoleAImp**-Objekt an seiner Schnittstelle die Operationen aller Rollen anbietet, muß bei jedem Aufruf einer derartigen Operation getestet werden, ob die zugehörige Rolle aktuell vorhanden ist. Für diese Aufgabe werden die internen `requireIsX`-Operationen zur Verfügung gestellt. Eine derartige Operation erzeugt eine Exception, wenn die Rolle nicht existiert. Die Operationen `makeX`, `isX` und `requireIsX` sind die ***Verwaltungsoperationen***, die für die Implementierung des Rollenmodells benötigt werden[36]. Bei einer Implementierung der Klasse **RoleAImp** muß festgelegt werden, wie zu verfahren ist, wenn eine `makeX`-Operation aufgerufen wird und bereits eine entsprechende Rolle vorhanden ist. Das Ergebnis des Aufrufs könnte eine Exception sein, alternativ wäre es auch denkbar, die alte Rolle durch die neue Rolle zu ersetzen.

Falls (mindestens) zwei Rollen eine Operation mit derselben Signatur, aber unterschiedlichen Realisierungen besitzen, wird die folgende Implementierung vorgeschlagen (`mS` sei die betroffene Operation in den Rollen **RoleX** und **RoleY**): (1) **RoleAImp** erhält zwei interne Operationen `mSX` und `mSY`. (2) Die nach außen sichtbare Operation `mS` testet, welche Rolle aktuell vorliegt und ruft die zugehörige interne Operation auf.[37]

[36] Konsequenterweise sollte es auch noch `deleteX`-Operationen geben, um eine Rolle wieder aufgeben zu können. Da diese in [Fow97b] nicht angegeben sind, wurden sie hier auch nicht aufgeführt.

[37] Was passiert, wenn gleichzeitig **RoleX**- und **RoleY**-Objekte vorliegen, ist in [Fow97b] nicht näher spezifiziert.

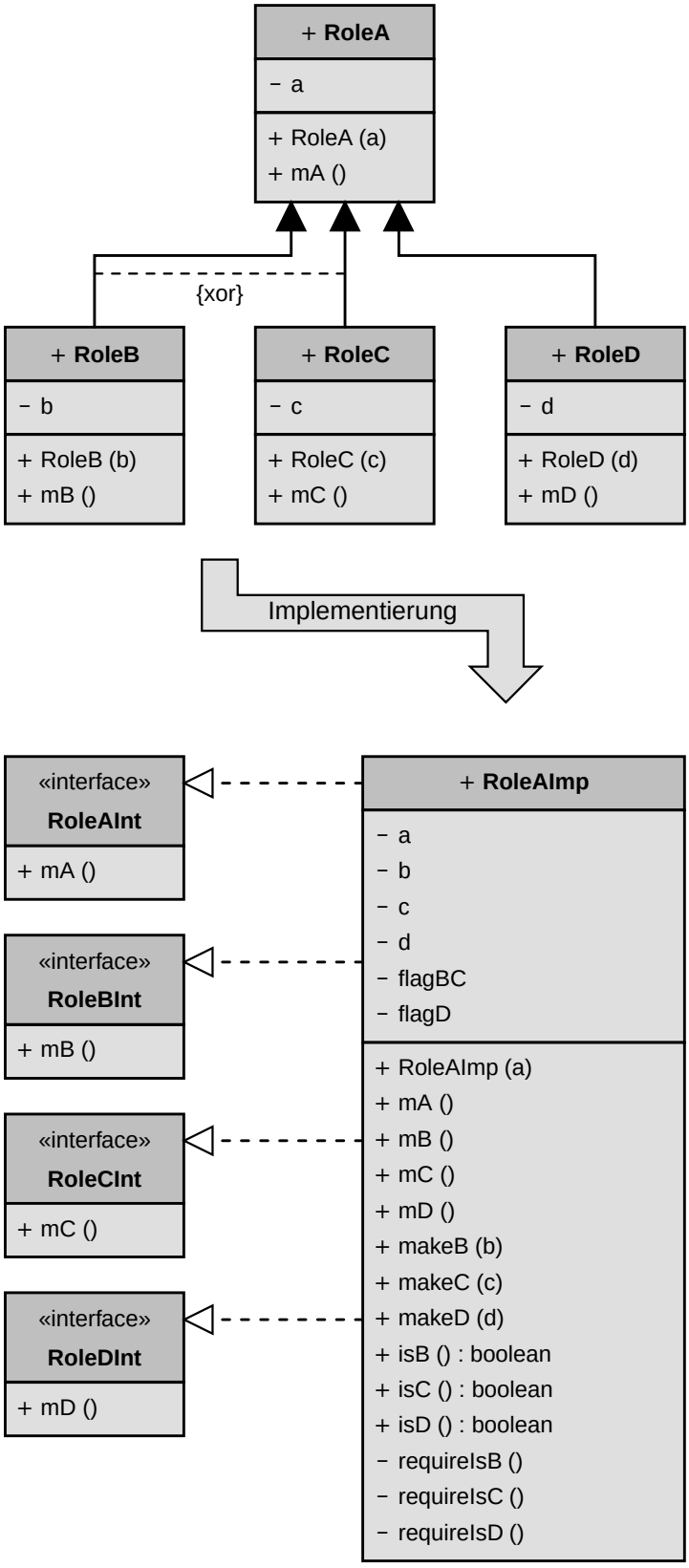


Abb. 26 Der Internal-Flag-Ansatz

Der Internal-Flag-Ansatz besitzt die folgenden Nachteile:

- Die **RoleAImp**-Klasse muß die Schnittstellen aller Unterrollen des Rollenmodells implementieren. Liegen sehr viele Rollen vor, entsteht eine sehr unübersichtliche Klasse.
- Analog enthält ein **RoleAImp**-Objekt die Attribute von allen Rollen, unabhängig davon, ob das **RoleAImp**-Objekt zur Zeit die Rollen überhaupt besitzt.
- Bei jedem Zugriff auf eine Operation, die nicht zu **RoleA** gehört, muß über die entsprechende `requireIs`-Operation zunächst getestet werden, ob der Zugriff überhaupt erlaubt ist. Da alle Rollen über ein einziges Objekt realisiert sind, besteht keine Möglichkeit, diese Aufgabe von der statischen Typprüfung des Übersetzers ausführen zu lassen. Selbst wenn man zwischen den Schnittstellen **RoleAInt** bis **RoleDInt** noch die Vererbungsstrukturen definieren würde, die durch das Rollenmodell vorgegeben sind (d.h. **RoleBInt**, **RoleCInt** und **RoleDInt** erweitern jeweils **RoleAInt**), müssen die `requireIs`-Operationen trotzdem verwendet werden. Es gibt keine Möglichkeit sicherzustellen, daß Zugriffe auf das **RoleAImp**-Objekt nur über Referenzen eines Schnittstellentyps stattfinden, da auch immer ein Zugriff über eine **RoleAImp**-Referenz möglich ist. Außerdem kann nicht garantiert werden, daß beim Zugriff über eine **RoleXInt**-Referenz das Objekt aktuell auch wirklich die **RoleX**-Rolle besitzt.
- Wenn in den Rollen Operationen mit derselben Signatur vorliegen, muß die Auswahl der Operation ausprogrammiert werden.

4.6.6 Der Hidden-Delegate-Ansatz

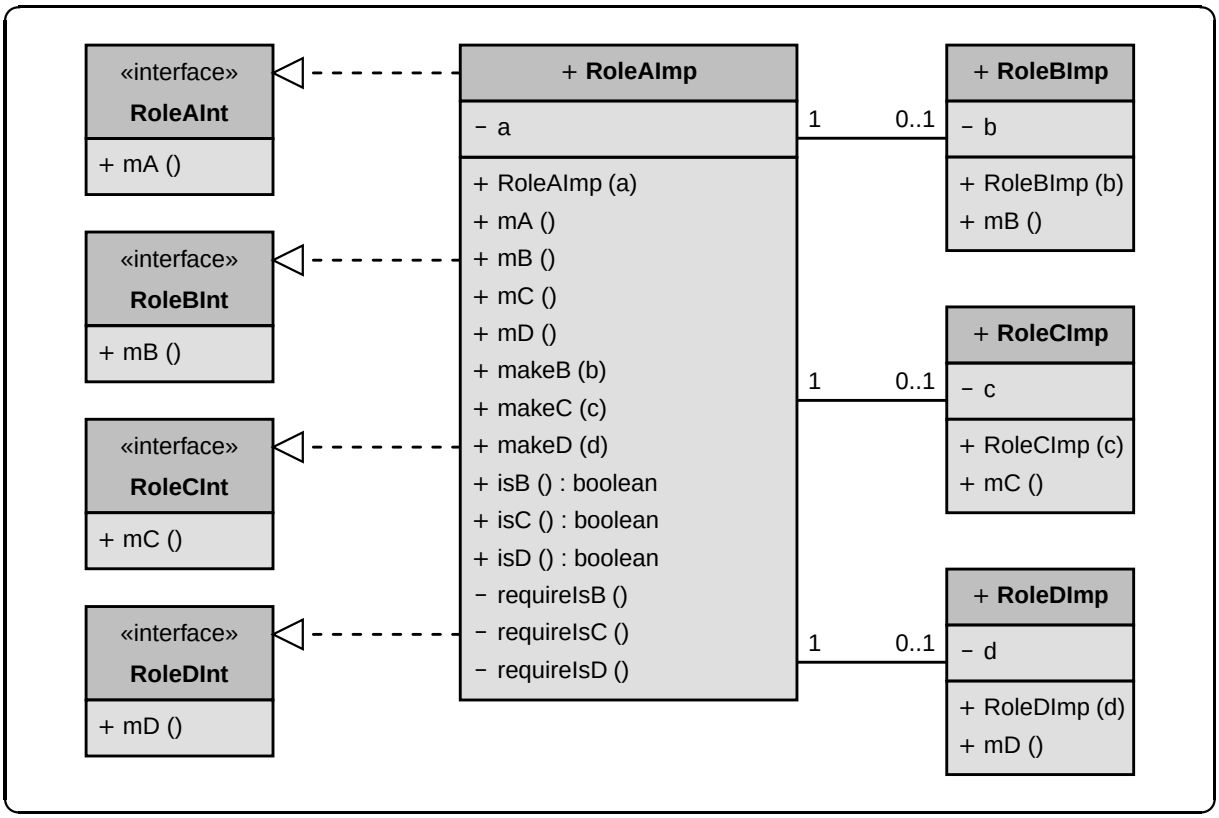


Abb. 27 Der Hidden-Delegate-Ansatz

Der Hidden-Delegate-Ansatz vermeidet den zweiten Nachteil der Internal-Flag-Lösung, indem er die Attribute und Operationen der einzelnen Rollen in eigene Klassen auslagert (vgl. auch [Fow97a, S. 284 ff]). Die Klasse **RoleAImp** implementiert aber weiterhin alle Schnittstellen des Rollenmodells sowie die erforderlichen Verwaltungsoperationen. In Abb. 27 (S. 63) ist die entstehende Implementierungsstruktur dargestellt, wobei das Rollenmodell aus Abb. 26 (S. 62) zugrunde gelegt wird. Ein Aufruf einer `makeX`-Operation führt zu der Erzeugung eines **RoleXImp**-Objekts. Diese Objekte sind aber für den Anwender des Rollenmodells nicht sichtbar. Der Aufruf einer Operation einer Rolle findet weiterhin über das **RoleAImp**-Objekt statt. Wenn der Aufruf erlaubt ist, wird er an das verdeckte Rollenobjekt delegiert. Somit bleiben die anderen Nachteile des Internal-Flag-Ansatzes auch beim Hidden-Delegate-Ansatz bestehen.

4.6.7 Der State-Object-Ansatz

Der State-Object-Ansatz basiert auf dem State-Muster ([GHJV95, S. 305-313], [DA98]) und läßt sich anwenden, wenn maximal eine Unterrolle angenommen werden kann, wie dies in Abb. 28 (S. 65) für die Rollen **RoleB**, **RoleC** und **RoleD** gilt. Die Schnittstelle zur Anwendung bildet analog zum Internal-Flag- und Hidden-Delegate-Ansatz die Klasse **RoleAImp**. Alle Aufrufe, die **RoleA** betreffen, werden von **RoleAImp** selbst behandelt. Aufrufe für eine Unterrolle werden dagegen an das assoziierte Objekt des **RoleBCDImp**-Typs delegiert. Nach der Erzeugung eines **RoleAImp**-Objekts besteht eine Beziehung zu einem **RoleBCDImp**-Objekt. Dieses Objekt liefert für einen Aufruf einer `isX`-Operation jeweils den Wert `false`, während für einen `mX`-Aufruf eine Exception über die interne Operation `incorrectTypeError` erzeugt wird. Die Aufgabe des **RoleBCDImp**-Objekts besteht somit in der Definition eines Standardfehlerverhaltens für den Fall, daß die gewünschte Rolle zur Zeit nicht vorhanden ist.[38] Eine `makeX`-Operation erzeugt ein Objekt der Klasse **RoleXImp** und ersetzt damit das Objekt des **RoleBCDImp**-Typs, das bisher mit dem **RoleAImp**-Objekt assoziiert war. Die **RoleXImp**-Klasse redefiniert genau die Operationen der **RoleBCDImp**-Klasse, die über die Rolle **RoleX** definiert sind. Ist es innerhalb der Ausführung einer **RoleXImp**-Operation erforderlich, auf die Funktionalität des zugehörigen **RoleAImp**-Objekts zuzugreifen, so kann dies über das Konzept der Selbstdelegation (*Self Delegation*, [Bec96]) erreicht werden (siehe Abb. 29 (S. 66)).
Gegenüber dem Internal-Flag- und dem Hidden-Delegate-Ansatz besteht hier der Vorteil, daß für Operationen mit derselben Signatur die Operationsauswahl nicht mehr ausprogrammiert werden muß. Stattdessen wird diese Aufgabe über den Mechanismus des dynamischen Bindens zur Laufzeit erledigt. Als wesentliche Nachteile des State-Object-Ansatzes lassen sich nennen:

- Eine Wurzelrolle kann maximal eine Unterrolle annehmen.
- Wie beim Internal-Flag- und Hidden-Delegate-Ansatz kann die Gültigkeit eines Operationsaufrufs nicht über eine statische Typprüfung kontrolliert werden.
- Die Implementierung der Wurzelrollenklasse wird unübersichtlich, wenn sehr viele Unterrollen im Rollenmodell spezifiziert wurden.

[38] In [Fow97b] wurde die zu **RoleBCDImp** analoge Klasse entsprechend dem State-Muster als abstrakte Klasse definiert. Damit tritt aber das zusätzliche Problem einer `null`-Pointer-Exception auf, wenn für das **RoleAImp**-Objekt zur Zeit keine Unterrolle definiert ist, also z.B. direkt nach der Erzeugung des **RoleAImp**-Objekts.

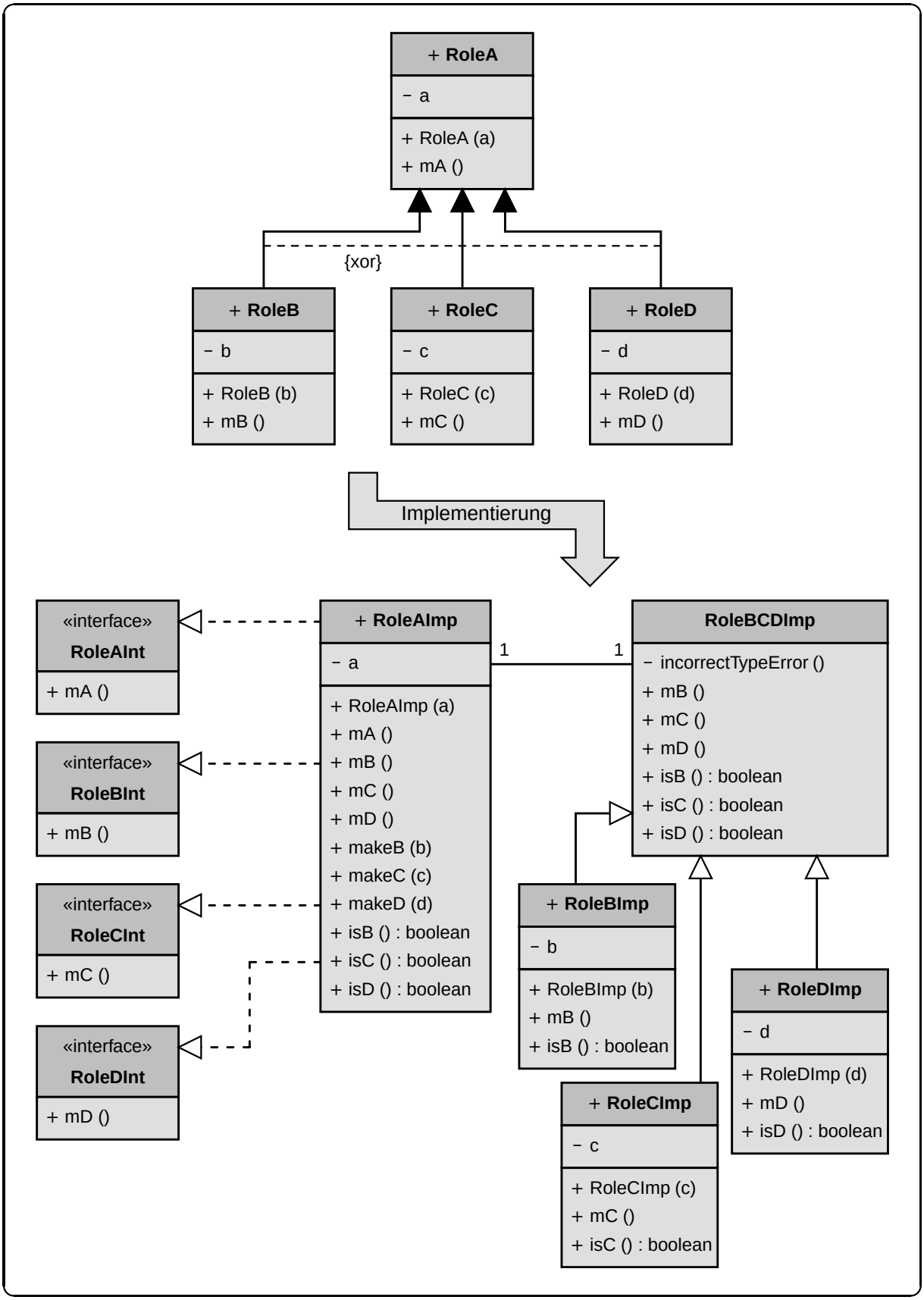
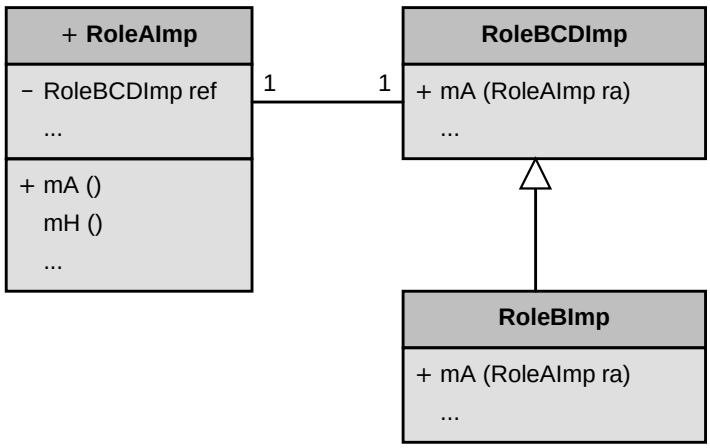


Abb. 28 Der State-Object-Ansatz



// in **RoleBImp**

```
public void mA (RoleAImp ra) {
   ra.mH();
   ...
}
```

// in **RoleAImp**

```
public void mA () {
   ref.mA (this);
   ...
}
```

Abb. 29 Das Konzept der Selbstdelegation

4.6.8 Der Role-Object-Ansatz

Das Role-Object-Muster wurde in [BRSW97] sowie [BRSW99] vorgestellt und besitzt die in Abb. 30 (S. 68) beschriebene Struktur[39]. Die Schnittstelle **RoleAInt** übernimmt die öffentlichen Operationen der Wurzelrolle **RoleA** und definiert die allgemeinen *Verwaltungsoperationen* eines Rollenmodells:

- `addRole(specification)`
  Der Aufruf dieser Operation erzeugt ein neues Rollenobjekt und fügt es zu der Rollenmenge hinzu, die von einem **RoleAImp**-Objekt verwaltet wird. Das Parameterobjekt `specification` legt fest, welche Rollenklasse betroffen ist, und stellt auch die notwendigen Intialisierungswerte für die Rollenobjekterzeugung bereit, z.B. einen Wert für `b`, wenn ein Objekt der Klasse **RoleBImp** erzeugt wird. Durch die Verwendung eines **Specification**-Objekts wird es möglich, die Menge der Rollentypen, die ein **RoleAImp**-Rollenobjekt prinzipiell verwalten kann, offen zu lassen. Die Implementierung von **RoleAImp** muß dann nicht geändert wer-

[39] Die Klassennamen aus [BRSW97] wurden an das in Abb. 30 (S. 68) verwendete Beispiel angepaßt. Es gilt die folgende Zuordnung: **Component** → **RoleAInt**; **ComponentCore** → **RoleAImp**; **ComponentRole** → **Role**; **ConcreteRoleA** → **RoleBImp**; **ConcreteRoleB** → **RoleCImp**.

den, wenn eine neue Unterrollenklasse benötigt wird, also beispielsweise die Rolle **RoleD** später in das Rollenmodell eingefügt wird. Für die Identifikation der zu erzeugenden Rolle kann im einfachsten Fall eine Zeichenkette verwendet werden. Innerhalb der **RoleAImp**-Klasse ist dann eine Abbildung der Zeichenkette auf die Klasse der zu erzeugenden Rolle notwendig. Als weitere Alternativen zur Identifikation, Verwaltung und Erzeugung von Rollenobjekten werden in [BRSW97] das *Product-Trader*-Muster ([BR98]), ein *Specification*-Muster ([EF97]) und das *Type-Object*-Muster ([JW98]) vorgeschlagen.

- `hasRole(specification)` liefert die Information zurück, ob ein Rollenobjekt des über `specification` definierten Typs bereits in **RoleAImp** vorhanden ist.
- `removeRole(specification)` löscht das über `specification` ausgewählte Rollenobjekt.
- `getRole(specification)` liefert eine Referenz auf das über `specification` defnierte Rollenobjekt zurück.

Den Zugriff auf eine Rolle erhält eine Anwendung, indem sie sich über einen `getRole`-Aufruf eine Referenz für das gewünschte Rollenobjekt besorgt. Alle Operationsaufrufe, welche sich auf die durch das Rollenobjekt neu definierte Funktionalität beziehen, werden von dem Rollenobjekt direkt ausgeführt (z.B. `mB` für ein **RoleBImp**-Objekt). Alle anderen Aufrufe (z.B. `mA`) werden an das zugehörige **RoleAImp**-Objekt delegiert, d.h. die entsprechenden Operationen der abstrakten Klasse **Role** leiten diese Aufrufe nur weiter. Ob dies auch für die Verwaltungsoperationen gilt, die ebenfalls auf einem Rollenobjekt aufrufbar sind, ist in [BRSW97] nicht näher spezifiziert. Konsequenterweise sollten die Aufrufe einer Verwaltungsoperation auf einem Rollenobjekt aber zur Auslösung einer Exception führen, da es einer Rolle nicht gestattet ist, für die direkte Oberrolle neue Rollen zu erzeugen oder zu löschen. Eine Lohnabrechnungsabteilung, die eine Person in der Rolle eines Angestellten verwaltet, darf für diese nicht einfach ein neues Projekt definieren, d.h. über die Angestelltenrolle dem Personenobjekt die Rolle eines Projektmitarbeiters zuordnen. Wenn dagegen der Angestelltenrolle weitere Unterrollen zugeordnet werden können, dann ist es notwendig, die Verwaltungsoperationen in der Angestelltenrolle zu redefinieren. Die Redefinition ist erforderlich, weil sich die Verwaltungsoperationen jetzt nicht mehr auf die Assoziation zwischen der **RoleAImp**- und der (abstrakten) **Role**-Klasse beziehen, sondern auf die Assoziation, die z.B. die **RoleBImp**-Klasse zu ihren Unterrrollen besitzt. Dies entspricht der in [BRSW97] beschriebenen rekursiven Anwendung des Role-Object-Musters.

Das Role-Object-Muster ist strukturell dem *Extension-Object*-Muster sehr ähnlich ([Gam98]): ein Objekt kann durch die Verwendung von *Extension*-Objekten um zusätzliche Funktionalitäten erweitert werden, wobei zum Definitionszeitpunkt des Objekts noch nicht bekannt ist, welche Erweiterungen später einmal notwendig sein werden. Der wesentliche Unterschied besteht darin, daß die zu **Role** korrespondierende Klasse **Extension** an ihrer Schnittstelle nicht die Operationen derjenigen Klasse enthält, deren Funktionalität ergänzt werden soll. Das Extension-Object-Muster wurde in [ZF98] und [Sch96] zur Definition von Rollenmodellen benutzt. Ein weiteres bezüglich der Struktur sehr ähnliches Muster ist das *Decorator*-Muster ([GHJV95, S. 175 ff]).

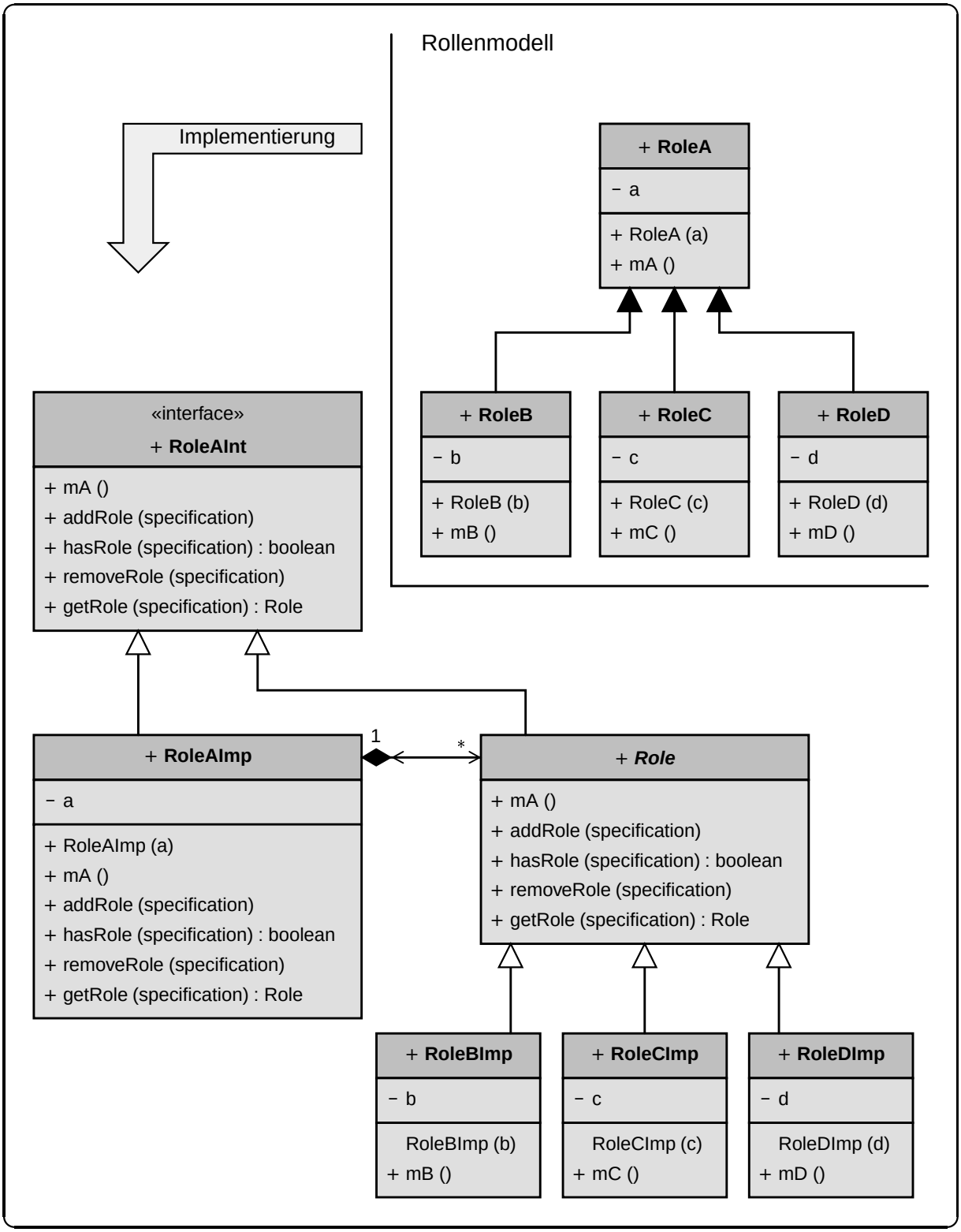


Abb. 30 Der Role-Object-Ansatz

4.7 Realisierungsansätze aus dem Programmiersprachenbereich

4.7.1 Der Extended-SMALLTALK-Ansatz

GOTTLOB, SCHREFL und RÖCK stellen in [GSR96] einen auf der Sprache SMALLTALK basierenden Ansatz zur Realisierung eines Rollenkonzepts vor. Das zugrunde liegende Rollenmodell ist eine Erweiterung der in [GKS90] und [SN88] beschriebenen Konzepte. Ein wesentliches Ziel war es, ein Rollenkonzept innerhalb einer bereits existierenden, weit verbreiteten objektorientierten Programmiersprache umzusetzen. Die Rollen eines Objekts bilden eine Baumstruktur, wobei eine Rolle alle Eigenschaften ihrer Oberrollen erbt. Die einzelnen Rollen können unabhängig voneinander dynamisch angenommen und wieder aufgegeben werden. Eine Rolle definiert eine eigenständige Sicht (Zugriffskontext) für das Gesamtobjekt (Objekt mit allen seinen Rollen). Ein Objekt kann eine Rolle mehrfach übernehmen.

Für die Umsetzung des Rollenkonzepts werden drei Klassen zur Verfügung gestellt (vgl. Abb. 31 (S. 70)):

- **ObjectWithRoles**
  Eine Klasse, welche die Wurzel einer Rollenhierarchie darstellen soll, muß **ObjectWithRoles** erweitern. Ein Objekt des Typs **ObjectWithRoles** verwaltet intern alle Rollen. Die Operation `getRole(roleType)` liefert die Referenz auf das Rollenobjekt zurück, welches über seinen Rollentyp identifiziert wird. Falls von einer Rolle mehrere Instanzen existieren, liefert `getQualifiedRole(qualifyingObject)` die entsprechende Referenz zurück, wobei `qualifyingObject` hier ein Objekt ist, über das die Rolleninstanz ausgewählt wird. Dieses Objekt wird beim Erzeugen der Rolle (siehe Klasse **RoleType**) als Parameter mitgegeben.
- **RoleType**
  Eine Rollenklasse, von der nur eine Rolleninstanz (pro **ObjectWithRoles**-Objekt) erzeugt werden soll, besitzt **RoleType** als Oberklasse. Über den Konstruktor `RoleType(objectWithRoles)` läßt sich eine Rolle erzeugen, die direkt dem Wurzelobjekt untergeordnet ist. `RoleType(role)` wird dagegen verwendet, wenn für ein Rollenobjekt `role` eine (direkte) Unterrolle zu erzeugen ist. `getRoot` liefert die Wurzel der aktuellen Rollenhierarchie, während `getAncestor` die Referenz auf die direkte Oberrolle zur Verfügung stellt. Der Aufruf von `delete` löscht die aktuelle Rolle und alle ihre Unterrollen.[40]
- **QualifiedRoleType**
  Alle Rollenklassen, von denen mehrere Instanzen erzeugbar sein sollen, müssen die Klasse **QualifiedRoleType** erweitern. In Abhängigkeit des verwendeten Konstruktors wird die neue Rolleninstanz entweder eine direkte Unterrolle des Wurzelobjekts oder eines anderen Rollenobjekts.

Die Methodensuche wird unter Verwendung der Delegation realisiert. Da SMALLTALK keine statische Typprüfung kennt, kann zur Laufzeit an ein Objekt eine beliebige Botschaft zur Aktivierung der zugehörigen Operation gesendet werden. Eine Botschaft, für die das Objekt keine Operation besitzt, wird an die vordefinierte Operation `doesNotUnderstand` zur Verarbeitung weitergereicht. Diese Operation wird von **RoleType** redefiniert. Eine nicht bekannte Bot-

[40] Da SMALLTALK keine Destruktoren bereitstellt, werden hier nur die internen Referenzen gelöscht.

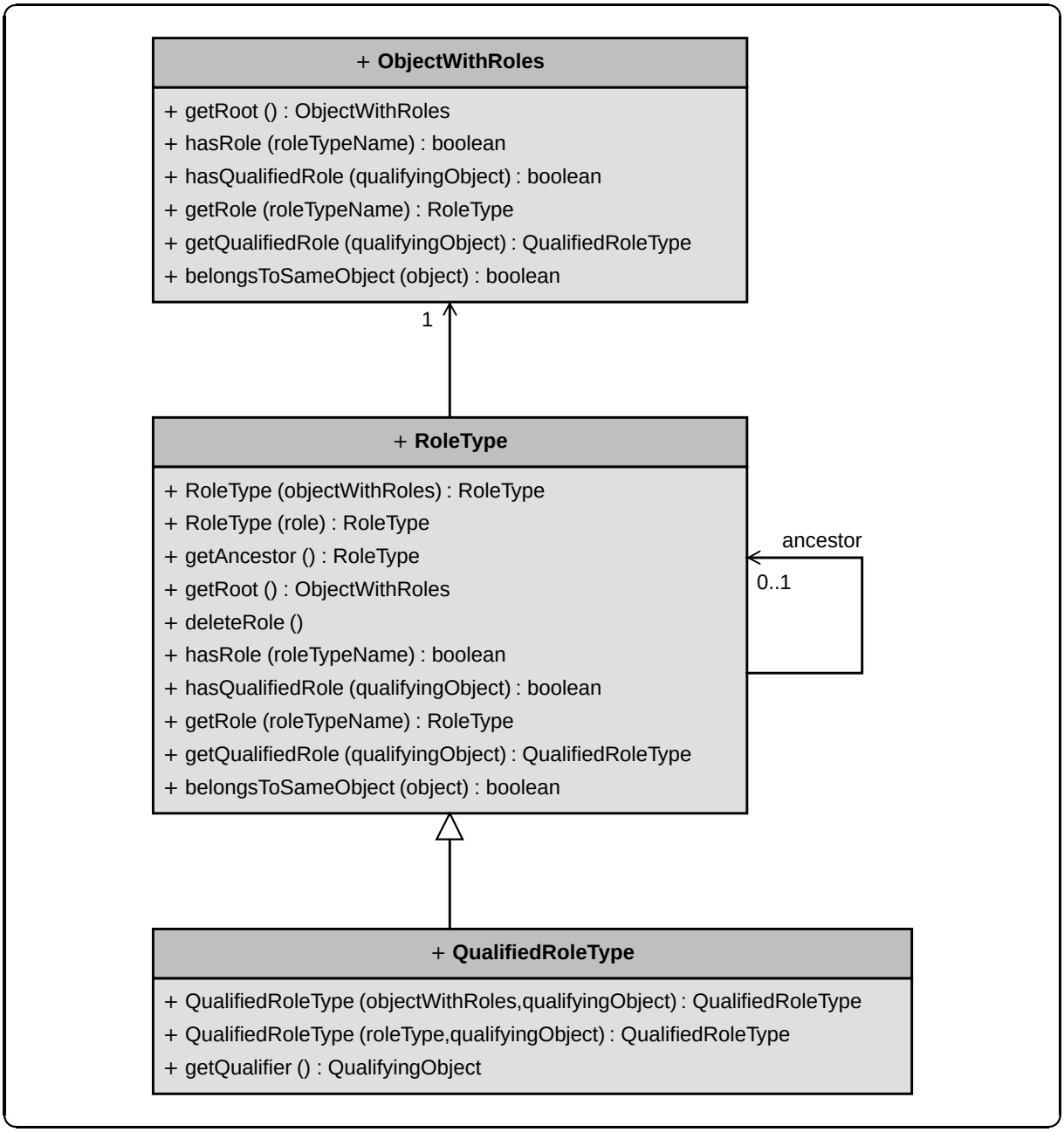


Abb. 31 Der Extended-SMALLTALK-Ansatz

schaft wird jetzt an die direkte Oberrolle weitergeleitet.

Der Extended-SMALLTALK-Ansatz besitzt die folgenden Nachteile:

- Durch die Verwendung der SMALLTALK-Sprache ist eine statische Typprüfung nicht möglich.
- Die Mehrfachvererbung von Rollen wird nicht unterstützt.
- Für Rollen können keine Restriktionen spezifiziert werden.

Das vorgestellte Rollenmodell ist nicht nur für den Bereich objektorientierter Programmiersprachen konzipiert worden, sondern auch Bestandteil einer Entwurfsumgebung für objektorientierte Datenbanken [KS91].

4.7.2 Das JAVA Role API

SCHREFL und THALHAMMER beschreiben in [ST01][41] eine JAVA-Umsetzung des Rollenkonzepts. Als Ausgangspunkt dient der für SMALLTALK in [GSR96] entwickelte Ansatz. Durch die in JAVA realisierte statische Typprüfung sind allerdings strukturelle Änderungen für eine Implementierung erforderlich. Die im JAVA Role Package zur Verfügung gestellten Klassen stellt Abb. 32 dar.[42]

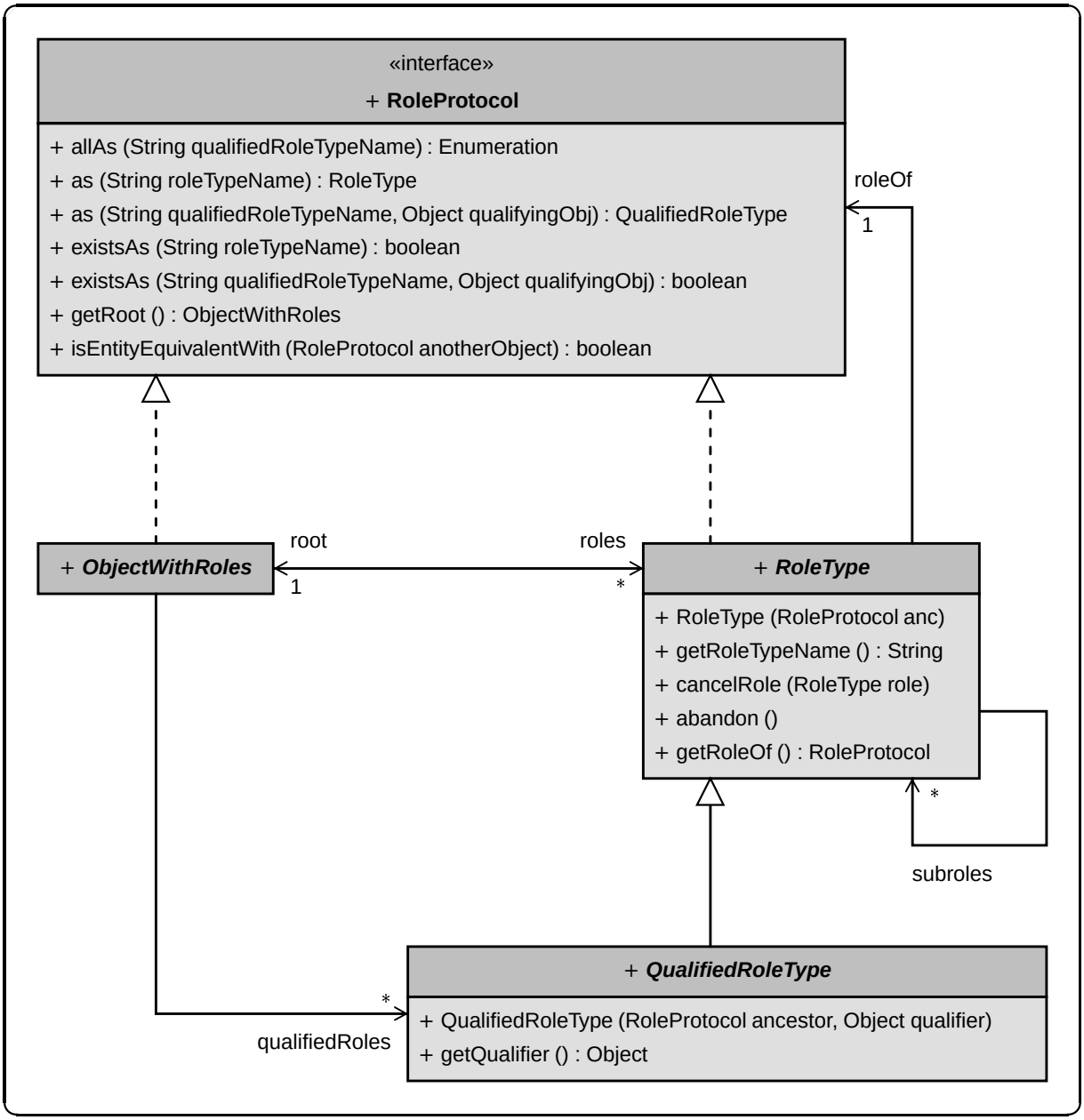


Abb. 32 Das JAVA Role Package

[41] Eine geringfügig korrigierte Version wurde in [ST03a] zur Veröffentlichung angenommen.

[42] Das JAVA Role Package kann von [ST03b] geladen werden. Für die einzelnen Klassen sind in Abb. 32 nur die **`public`**-Methoden angegeben, da nur auf diese in [ST01] Bezug genommen wird.

Die abstrakte Klasse **ObjectWithRoles** wird als Oberklasse für eine Wurzelklasse einer Rollenhierarchie verwendet. Klassen, die die Rolleneigenschaft übernehmen sollen, werden von den abstrakten Klassen **RoleType** bzw. **QualifiedRoleType** abgeleitet. **RoleType** ist der Ausgangspunkt für Rollen, von denen maximal eine Instanz erzeugbar sein soll, während von einer Rolle, die **QualifiedRoleType** erweitert, mehrere Rolleninstanzen existieren können. Das Interface **RoleProtocol** definiert das gemeinsame Verhalten dieser drei Klassen.

- **RoleProtocol**
  Die Operation `allAs` liefert alle Unterrollenreferenzen eines bestimmten Typs zurück. Dagegen werden `as(roleTypeName)` und `as(qualifiedRoleTypeName,qualifyingObj)` verwendet, um eine einzelne Referenz eines Unterrollentyps als Ergebnis zu erhalten. Die beiden `existsAs`-Operationen liefern die Information, ob eine Unterrolleninstanz eines bestimmten Typs vorhanden ist. `getRoot` stellt eine Referenz auf das Wurzelrollenobjekt der Rollenhierarchie zur Verfügung. Mittels `isEntityEquivalentWith(anotherObject)` läßt sich prüfen, ob zwei Rolleninstanzen derselben Wurzelrolleninstanz zugeordnet sind.
- **RoleType**
  Der Konstruktor besitzt als Parameter die Referenz auf die direkte Oberrolle. Damit wird die *roleOf*-Beziehung hergestellt. Die Operation `getRoleTypeName` liefert den voll qualifizierten Klassennamen des Rollentyps als Ergebnis. `cancelRole(role)` löscht aus der Menge der direkten Unterrollen die Instanz `role`. Über die Operation `abandon` wird die aktuelle Rolleninstanz gelöscht. Durch den Aufruf werden auch alle Unterrolleninstanzen gelöscht. Ein Aufruf der Operation `getRoleOf` liefert die Referenz auf die direkte Oberrolle, die vom Typ `RoleType` oder `ObjectWithRoles` sein kann.
- **QualifiedRoleType**
  Der Konstruktor besitzt neben einer Referenz auf die direkte Oberrolle den Parameter `qualifier`. Die `qualifier`-Werte werden verwendet, um die einzelnen Rolleninstanzen dieses Typs, die einer (gemeinsamen) Oberrolle zugeordnet sind, eindeutig identifizieren zu können. Die Operation `getQualifier` liefert das `qualifier`-Objekt für die Unterrolle als Ergebnis.

Eine Klasse **R**, die von **RoleType** abgeleitet wird, muß einen Konstruktor definieren, der als Parameter eine Referenz auf die direkte Oberrollenklasse besitzt. Liegt z.B. als Wurzelrolle die Klasse **Person** vor, die folglich **ObjectWithRoles** erweitert, dann ist für eine Klasse **Student** ein Konstruktor bereitzustellen, der einen Parameter des Typs `Person` definiert. Damit wird sichergestellt, daß eine Studentenrolle nur ein Personenobjekt als direkte Oberrolle besitzen kann. Entsprechend gilt, daß eine Klasse **QR**, die **QualifiedRoleType** erweitert, einen Konstruktor haben muß, der als Parameter eine Referenz auf die direkte Oberrollenklasse hat. Zusätzlich sollte er einen Parameter besitzen, der die eindeutige Identifizierung der direkten Unterrolleninstanzen einer Oberrolleninstanz erlaubt.

Die vorgestellte Struktur zur Umsetzung des Rollenkonzepts besitzt die folgenden Nachteile:

- Da JAVA nur Einfachvererbung kennt, wird eine Mehrfachvererbung von Rollen nicht unterstützt.
- JAVA verfügt zwar im Gegensatz zu SMALLTALK über eine statische Typprüfung, die spezifizierten Operationen zum Ermitteln der Referenzen von Rolleninstanzen (`allAs`, `as`,

`getRoot` und `getRoleOf`) liefern allerdings Referenzen zurück, die im Programm erst durch Anwendung des Cast-Operators in den Originaltyp der Rolleninstanz umgewandelt werden müssen. Die semantische Korrektheit der Anwendung des Cast-Operators kann vom Übersetzer nicht kontrolliert werden, so daß es bei einer fehlerhaften Nutzung zu Laufzeitfehlern und damit zum Programmabbruch kommen kann.

- Für Rollen lassen sich keine Restriktionen formulieren.
- Es ist ungeklärt, wie sich Rolleninstanzen verhalten, die über `abandon` bzw. `cancelRole` gelöscht wurden. Da JAVA keine Destruktoren kennt und es keinen Befehl gibt, alle Referenzen auf eine Rolleninstanz zu löschen, bliebe nur die Möglichkeit, die Rolleninstanz als gelöscht zu markieren und alle dann folgenden Zugriffe z.B. mit einer entsprechenden Exception zu beantworten.[43]
- Über eine Unterrollenreferenz ist es nicht möglich, auf die Operationen der Oberrollen zuzugreifen. Damit ist die *roleOf*-Beziehung semantisch nicht realisiert.
- Falls eine Klasse **AA**, die selbst schon eine andere Klasse **A** erweitert, den Ausgangspunkt einer Rollenhierarchie bilden soll, geht dies nicht mehr durch die Erweiterung von **ObjectWithRoles**, da JAVA auf Klassenebene nur Einfachvererbung unterstützt. Hierfür wird im JAVA Role API die Klasse **ProxyObjectWithRoles** angeboten. Um diese Klasse zu verwenden, muß die Klasse **AA** die Schnittstelle **RoleProtocol** implementieren. Zu diesem Zweck wird ein Objekt der Klasse **ProxyObjectWithRoles** erzeugt. Für jede Operation aus **RoleProtocol** erfolgt eine Delegation an die entsprechende Operation des **ProxyObjectWithRoles**-Objekts (vgl. Abb. 33 (S. 74)).
  Der wesentliche Nachteil dieser Lösung liegt in der Notwendigkeit, jedesmal wieder unter Verwendung der Delegation das **RoleProtocol** Interface implementieren zu müssen.
  Für ein Klasse, die die Funktionalität einer Rolle übernehmen soll, besteht dasselbe Problem, falls sie bereits eine andere Klasse erweitert. Allerdings stellt das JAVA Role API weder eine **ProxyRoleType** noch eine **ProxyQualifiedRoleType**-Klasse zur Verfügung.

[43] Da weder `abandon` noch `cancelRole` aus [ST03b] eine Wirkung zeigen, bleibt die beabsichtigte Semantik der Löschoperation unbeantwortet. Ein logisches Löschen einer Rolleninstanz scheint es aber nicht zu geben. Eine entsprechende Exception-Klasse ist in dem JAVA Role API nicht definiert.

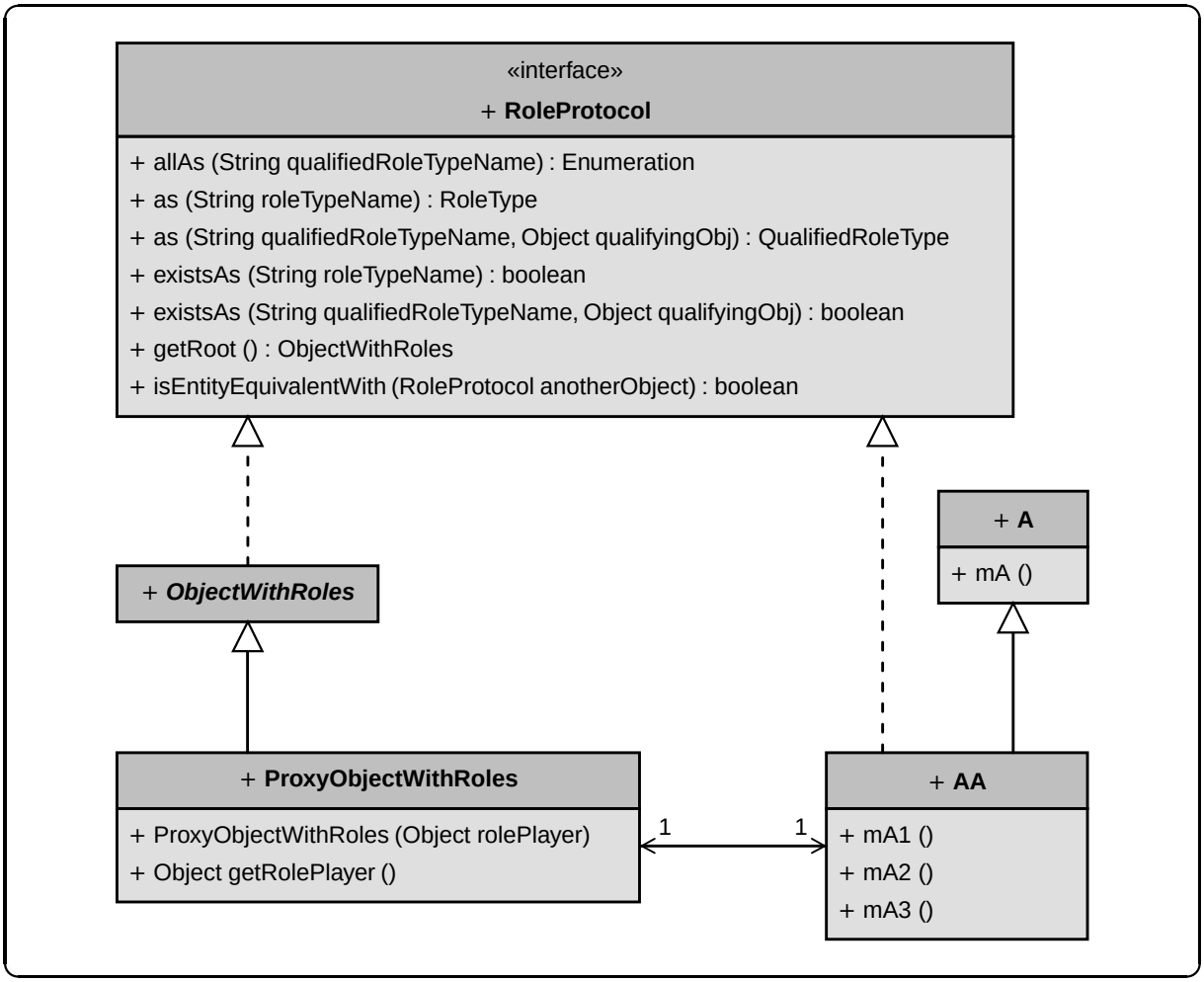


Abb. 33 Verwendung der Klasse **ProxyObjectWithRoles**

4.7.3 Die Anwendung der Aspektorientierung

Grundlagen

Das Konzept der **aspektorientierten Programmierung** (*Aspect-Oriented Programming*, AOP) wurde erstmalig von KICZALES et al. in [KIL+97] vorgestellt. AOP ist eine Technologie, die es erleichtern soll, eine **Trennung von Zuständigkeiten** (*Separation of Concerns*, vgl. [Par72] und [Par75]) innerhalb der Software zu erreichen. Das Konzept basiert auf der Beobachtung, daß viele Programme die folgende Struktur aufweisen:

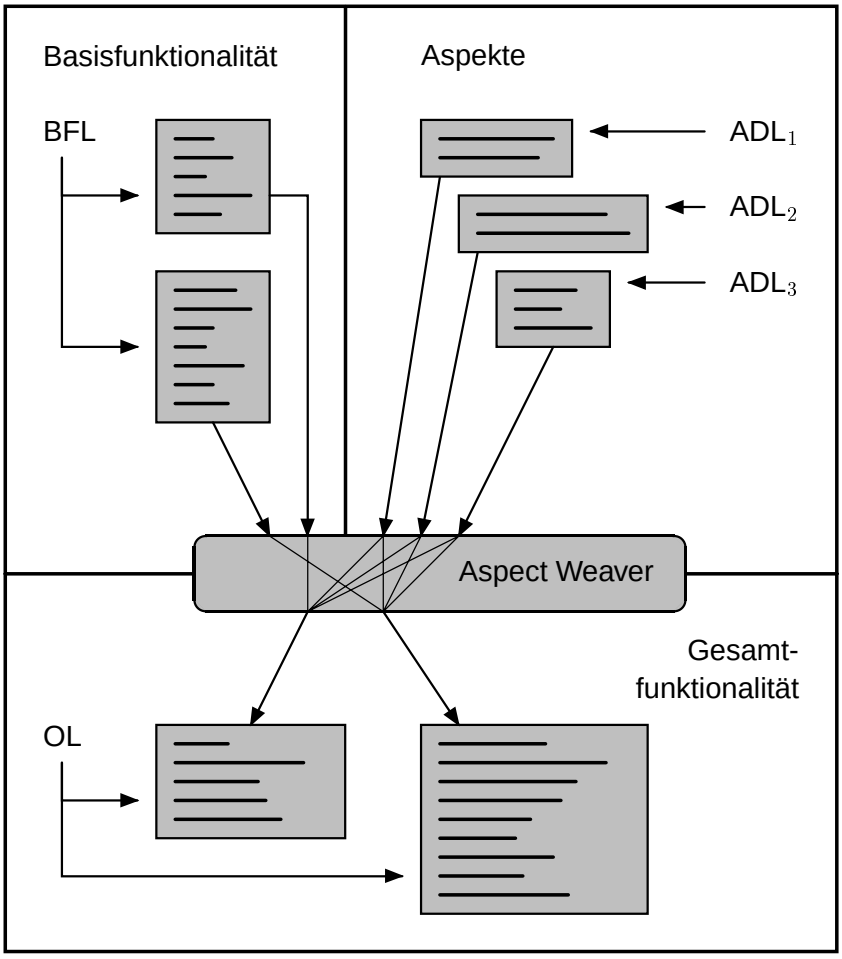
- Es gibt eine gewisse Grundfunktionalität.
- Neben der Grundfunktionalität enthält das Programm bestimmte Aspekte, wie zum Beispiel den Synchronisationsaspekt oder den Fehlerbehandlungsaspekt. Der zu den Aspekten gehörende Quelltext ist über das gesamte Programm verstreut. In [KHH+01] wird in diesem Zusammenhang auch von einem **Cross-Cutting Concern** gesprochen.

Dieses Programmiermodell hat verschiedene Nachteile:

- Durch die Verteilung des Quelltextes eines Aspekts über das gesamte Programm ist es sehr schwierig, diesen Quelltext in dem Programm zu lokalisieren, um den Aspekt aus dem Programm zu ändern oder zu entfernen. Wird beispielsweise ein Programm, das für eine Mehrprozeßumgebung geschrieben wurde, in einer Einprozeßumgebung eingesetzt, so ist der Synchronisationsaspekt überflüssig und könnte prinzipiell entfernt werden.
- Umgekehrt kann es sehr aufwendig sein, einen neuen Aspekt in ein bestehendes Programm zu integrieren. Soll zum Beispiel während der Testphase eines Programms für eine Methode `m1` protokolliert werden, von welchen anderen Methoden aus sie aufgerufen wird, müssen alle Aufrufstellen von `m1` lokalisiert werden, um dort entsprechende Ausgabeanweisungen einzubauen.[44]
- Zusätzlich kann das Problem auftreten, daß sich mehrere Aspekte eines Programms gegenseitig beeinflussen (*Tangling-Of-Aspects-Phänomen*, vgl. [KIL+97]). Hat man beispielsweise einen Synchronisationsaspekt und einen Tracing-Aspekt, dann wird die Aufgabe des Tracing-Aspekts auch darin bestehen, die Aufrufe von Methoden des Synchronisationsaspekts zu protokollieren.

Um eine Entkopplung der Aspekte von der Basisfunktionalität eines Programms zu erreichen, wird die in Abb. 34 (S. 76) dargestellte Struktur vorgeschlagen. Die Basisfunktionalität des Programms wird in einer dafür geeigneten Programmiersprache (*Basic Functionality Language*, BFL) beschrieben. Die Formulierung der einzelnen Aspekte kann unter Verwendung eigener Programmiersprachen (*Aspect Description Language*, ADL) stattfinden. Damit läßt sich auch das Problem lösen, daß Standardprogrammiersprachen für bestimmte Aspekte keine geeigneten Sprachkonstrukte anbieten. So gibt es beispielsweise in der Sprache C keine direkte, in die Sprache integrierte Unterstützung eines Synchronisationskonzepts. Die Basisfunktionalität und die Aspekte dienen als Eingabe für den Aspect Weaver. Dieser erzeugt ein neues Programm in der Ausgabesprache OL, die nicht notwendigerweise mit der BFL übereinstimmen muß.

[44] Dies ist eine Funktionalität, die ein Debugger anbieten könnte. Oft besitzt der verwendete Debugger aber die für den aktuellen Test benötigte Funktionalität nicht, so daß der Anwendungsprogrammierer entsprechenden Test- oder Protokoll-Code in sein Programm *hineinprogrammieren* muß.



BFL: Basic Functionality Language
ADL: Aspect Description Language
OL: Output Language

Abb. 34 Das Konzept der aspektorientierten Programmierung

AspectJ

Die Kombination aus JAVA und ASPECTJ stellt eine Möglichkeit zur Anwendung der Aspektorientierung dar. JAVA übernimmt die Rollen der BFL und OL, während ASPECTJ zur Formulierung der Aspekte dient. ASPECTJ ist dem ECLIPSE-Projekt untergliedert, das eine *universelle Tool-Plattform* bereitstellt ([ecl03])[45].

Eine ausführliche Einführung in den ASPECTJ-Sprachumfang findet sich in [The03a]. An dieser Stelle wird nur auf das grundlegende Konzept eingegangen. Hierfür sind vier zentrale Begriffe zu definieren:

- **Join Point**
  Ein Join Point repräsentiert eine prinzipielle Stelle im Kontrollfluß des Programms. Beispiele

[45] Die jeweils aktuelle ASPECTJ-Version kann von `http://www.eclipse.org/aspectj` bezogen werden.

für Join Points sind: Methodenaufruf, Methodenausführung, Objektinstanziierung, Konstruktorausführung und Attributzugriff.

- **Pointcut**
  Über einen Pointcut werden konkrete Join Points ausgewählt. Der Pointcut `call(void Point.set*(int))` wählt beispielsweise alle Methodenaufrufe aus, für die folgende Eigenschaften gelten: Die Methode hat den Ergebniswert `void`, gehört zur Klasse `Point`, der Methodenname beginnt mit `set` und die Methode besitzt genau einen formalen Parameter des Typs `int`.
- **Advice**
  Ein Advice ordnet einem Pointcut die Funktionalität zu, die auszuführen ist, wenn während des Programmablaufs einer der zugehörigen Join Points erreicht wird. Zusätzlich definiert der Advice den genauen zeitlichen Ablauf: ein *Before Advice* wird aktiviert, wenn der Join Point erreicht wurde, ein *After Advice*, nachdem der Join Point verarbeitet wurde. Ein *Around Advice* wird statt des Join Points ausgeführt. Der folgende ASPECTJ-Programmausschnitt definiert zunächst einen *benannten* Pointcut `move`, der fünf Join Points enthält. Der folgende *Before Advice* legt fest, daß vor der Ausführung einer der entsprechenden Operationen eine Ausgabeanweisung erfolgt.

```
pointcut move():
  call ( void FigureElement.setXY(int,int) ) ||
  call ( void Point.setX(int) )   ||
  call ( void Point.setY(int) )   ||
  call ( void Line.setP1(Point) ) ||
  call ( void Line.setP2(Point) );

  before(): move() {
    System.out.println ( "about to move" );
  }
```

- **Aspect**
  Aspect ist die syntaktische Einheit, die Pointcut- und Advice-Definitionen aufnimmt. Zusätzlich können innerhalb eines Aspekts für eine oder mehrere Klassen neue Attribute, Konstruktoren und Methoden deklariert werden. Diese Deklarationen werden als **Inter-type Declarations** bezeichnet. Das folgende Beispiel definiert für eine Klasse `Point` eine neue Methode innerhalb des Aspekts `PointAssertion`. Diese Methode kontrolliert vor jedem Aufruf einer `setX`-Methode auf einem `Point`-Objekt, ob der neue `x`-Wert zwischen `0` und `100` liegt.

```
class Point {
  private int x, y;
  public void setX(int x) { this.x=x; }
  public void setY(int y) { this.y=y; }
}
aspect PointAssertion {
  private boolean Point.assertX(int x) {
    return x>=0 && x<=100;
  }
```

```
    before(Point p, int x) : target(p) && args(x)
                             && call(void setX(int)) {
      if ( !p.assertX(x)) {
         System.out.println ( "Illegal value for x" );
         return;
      }
    }
  }
```

Durch die Parametrisierung der `before`-Klausel ist innerhalb des Advices der Zugriff auf das von dem `setX`-Aufruf betroffene `Point`-Objekt und den Parameter `x` möglich. Wäre `assertX` aus dem obigen Beispiel nicht als `private` sondern als `public` deklariert worden, so könnte über eine Referenz des Typs `Point` auf diese Methode zugegriffen werden. Neben den Inter-type Deklarationen kann ein Aspekt auch eigene Attribute und Methoden deklarieren. Diese sind aber nur innerhalb des Aspekts nutzbar, ein Zugriff über eine Objektreferenz ist nicht möglich.

Realisierung von Rollenmodellen mit AspectJ – der Hybrid-Ansatz

Für die Umsetzung von Rollenmodellen mit ASPECTJ werden von KENDALL zwei Entwurfsansätze präsentiert ([Ken99b], [Ken00a]). Da beide auf der ASPECTJ-Version 0.2 basieren, werden hier kurz einige wichtige Unterschiede zu der oben beschriebenen aktuellen ASPECTJ-Version vorgestellt:

- Ein Aspekt kann explizit mit dem `new`-Operator erzeugt werden.
- Durch die auf Aspektinstanzen definierte Methode `addObject` läßt sich einem Aspekt zur Laufzeit ein Objekt zuordnen. Damit erfolgt die Aktivierung aller Advices, die für die Klasse des Objekts innerhalb des Aspekts definiert wurden.
- Durch den Aufruf der Methode `removeObject` werden das Objekt und die Aspektinstanz wieder getrennt.
- Die Methode `getObjects` liefert als Ergebnis ein `Vector`-Objekt, das Referenzen auf alle Objekte enthält, die aktuell der Aspektinstanz zugeordnet sind.
- In der Klasse `Object` gibt es die neue Methode `getAspects`, die ein `Vector`-Objekt mit den Referenzen aller Aspektinstanzen zurückliefert.

Die beiden Ansätze werden in [Ken99b] anhand des Rollenmodells des BUREAUCRACY-Musters ([Rie98], [Rie97]) erläutert (vgl. Abb. 35 (S. 79)). Der erste Realisierungsansatz wird als **Hybrid-Ansatz** bezeichnet und besitzt die folgende Struktur:

- Objekte der Klasse `Agent` übernehmen die verschiedenen Rollen.
- Für jede Rolle wird ein eigener Aspekt definiert.
  - Der Aspekt besitzt entsprechende Attribute und Operationen, um die Beziehungen zu anderen Rollen herzustellen. Beispielsweise besitzt der Aspekt `Client` eine Methode `setClerk(Clerk c)`, um die Beziehung zu einer `Clerk`-Rolle aufzubauen.[46]

[46] Die in [Ken99b] angegebene Implementierung des `Manager`-Aspekts erlaubt nur den Aufbau einer Beziehung zu einer `Subordinate`-Rolle, obwohl die spezifizierte Kardinalität auch Werte größer als 1 zuläßt.

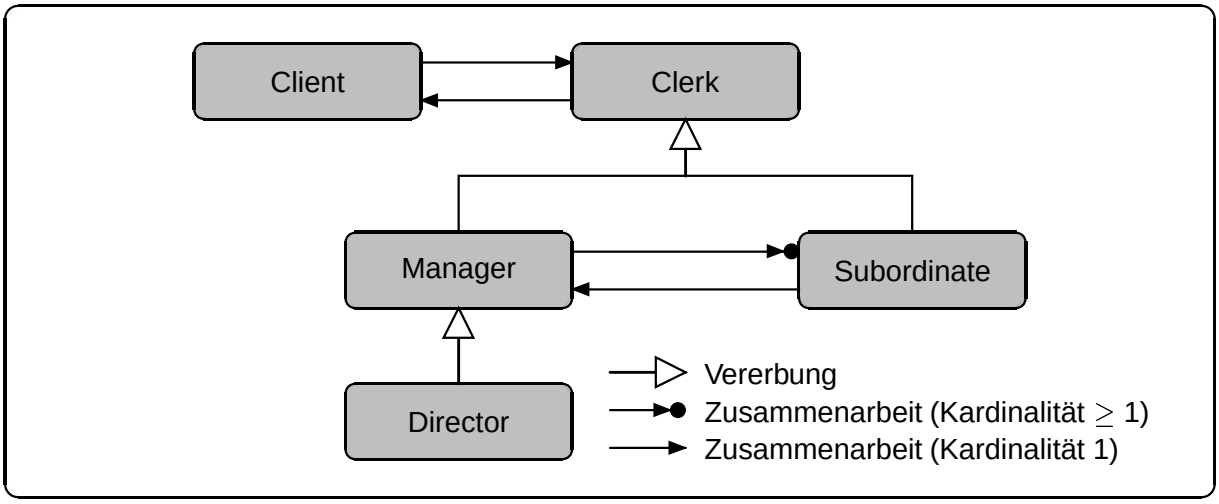


Abb. 35 Das Rollenmodell des BUREAUCRACY-Musters

- Falls die Rolle einen eigenen Zustand (Rollenkontext) besitzt, wird dieser über Attribute innerhalb des Aspekts realisiert.
- Der Aspekt deklariert Inner-type Methoden für die `Agent`-Klasse, wobei der Methodenrumpf leer bleibt. Wird ein `Agent`-Objekt erzeugt, so besitzt es die Schnittstelle aller Aspekte, der Aufruf einer entsprechenden Methode bleibt aber wirkungslos, solange das Objekt noch nicht einer entsprechenden Aspektinstanz zugeordnet wurde.[47]
- Die eigentliche Funktionalität der Rollenmethoden wird über Advices im Aspekt definiert. Für jede Inner-type-Methode des Aspekts existiert dann ein `before`-Advise.

Die Vererbungsbeziehungen zwischen den Aspekten entsprechen denen des Rollenmodells. Daher erweiterten die Aspekte `Manager` und `Subordinate` den Aspekt `Clerk`. Zusätzlich erweitert der `Director`-Aspekt den `Manager`-Aspekt.[48]

Da auf die rollenspezifischen Methoden über eine `Agent`-Referenz zugegriffen wird, lassen sich auf diesem Weg mehrere Instanzen einer Rolle desselben Typs nicht unterscheiden. Daher wird der obige Ansatz für diesen Fall um einen Rollenkontext ergänzt:

- Alle Aspekte erweitern den Aspekt `Role`. Dieser enthält ein Attribut `context` (z.B. des Typs `String`).
- Alle Methoden eines Aspekts erhalten einen zusätzlichen `context`-Parameter.
- Der Aufruf einer derartigen Methode aktiviert den zugehörigen Advise aller Aspektinstanzen, die dem Objekt zugeordnet sind.
- Der Advise prüft, ob der Wert des `context`-Parameters mit dem Wert des rolleninternen `context`-Attributs übereinstimmt. Im Erfolgsfall wird der Advise weiter ausgeführt, andernfalls sofort beendet, was bedeutet, daß der Methodenaufruf für diese Aspektinstanz wirkungslos bleibt.

[47] Als Alternative zu einem leeren Methodenrumpf kann die Inner-type-Methode auch eine Exception erzeugen.

[48] Bei der angegebenen Implementierung erweitern `Clerk` und `Client` noch den Aspekt `Role`. Allerdings enthält [Ken99b] keine Informationen über die Funktionalität von `Role`. Der in [Ken98] angegebene Verweis `www.cs.rmit.edu.au/~kendall` für den vollständigen Quelltext existiert nicht mehr und auf der aktuellen Homepage von E. KENDALL (`www.pscit.monash.edu.au/~kendall`) befindet sich kein Verweis auf den Quelltext.

Der Glue-Ansatz

Ein Aspekt des **Hybrid**-Ansatzes hat die zentralen Eigenschaften, daß er einerseits über seine Advises die Funktionalität der rollenspezifischen Methoden bereitstellt, und andererseits über die Inner-type-Deklarationen den expliziten Bezug zur `Agent`-Klasse realisiert. Im sogenannten **Glue**-Ansatz werden diese beiden Punkte entkoppelt:

- Für jede Rolle wird eine eigene Klasse zur Verfügung gestellt. Diese implementiert das rollenspezifische Verhalten.
- Diese Rollenklassen besitzen diesselbe Vererbungshierarchie wie die Rollen des Rollenmodells.
- Jeder Rollenklasse ist zusätzlich ein eigener Aspekt zugeordnet.
  - Der `Role`-Aspekt besitzt Inner-type-Deklarationen für den Aufbau der Beziehung zwischen dem `Agent`-Objekt und einer Rolleninstanz.
  - Die verbleibenden Aspekte definieren für die Methoden einer Rolle jeweils eine Inner-type-Methode, die der `Agent`-Klasse zugeordnet wird. Ein Aufruf dieser Methode für ein `Agent`-Objekt delegiert den Kontrollfluß an die entsprechende Methode des Rollenobjekts, sofern dieses dem `Agent`-Objekt aktuell zugeordnet ist.

Nachteile der AspectJ-Ansätze

Beide Ansätze besitzen die folgenden Nachteile:

- Es werden keine Restriktionen unterstützt. Alle Restriktionen für Rollen sind direkt auszuprogrammieren.
- Die Verarbeitung von Methodenaufrufen für Rollen mit einer Kardinalität größer als 1 ist sehr aufwendig, da für jede Rolleninstanz zunächst überprüft wird, ob sie überhaupt von dem Methodenaufruf betroffen ist.
- Da weder JAVA (für Klassen) noch ASPECTJ (für Aspekte) Mehrfachvererbung anbieten, kann eine Rolle nicht mehrere direkte Oberrollen besitzen.
- Eine Rolle selbst kann keine weiteren Rollen übernehmen. Die Rolleninstanzen sind alle an die `Agent`-Objektinstanz gebunden.
- Rolleninstanzen können zwar dynamisch einem Objekt zugeordnet werden, die Methodenschnittstelle aller Rollen ist aber statisch in der `Agent`-Klasse vorhanden. Damit kann immer auf eine Rolleninstanz zugegriffen werden, selbst wenn das `Agent`-Objekt diese noch gar nicht explizit übernommen hat.
- Da immer über eine `Agent`-Referenz auf die rollenspezifischen Methoden zugegriffen wird, besitzen alle Nutzer des `Agent`-Objekts die vollständige Gesamtobjektsicht, d.h. sie können auf alle vorhandenen Rolleninstanzen zugreifen.
- Die Komposition von Rollen wird nur ansatzweise unterstützt, da ASPECTJ nicht über die notwendigen Eigenschaften verfügt (vgl. [Ken99b, S. 364]). Ein Problem tritt z.B. dann auf, wenn ein `Agent`-Objekt Rolleninstanzen aus unterschiedlichen Rollenmodellen annehmen soll. Besitzen die Rollen Methoden mit derselben Signatur, so ist ein Verschmelzen dieser beiden Rollen sinnvoll.[49] Dieses Verschmelzen würde hier bedeuten, daß nur eine der

[49] Eine Voraussetzung ist natürlich, daß auch die Semantik der Methoden identisch ist.

beiden Methoden der `Agent`-Klasse zugeordnet wird. Der ASPECTJ Compiler erlaubt es jedoch nicht, zwei Aspekte zu verwenden, die dieselbe Inner-type-Deklaration für eine bestimmte Klasse besitzen. Für die Erzeugung des zusammengesetzten Rollenmodells ist es daher erforderlich, zumindest einen der beiden Aspekte anzupassen, indem die Deklaration der Inner-type-Methode herausgenommen wird.

Beim **Glue**-Ansatz tritt zusätzlich das Problem der Objektschizophrenie (*Object Schizophrenia*) auf (vgl. [IBM], [SR02]), da die Gesamtfunktionalität eines Objekts, welches k Rollen besitzt, über $k+1$ Objekte verteilt ist.

Da es seit der ASPECTJ-Version 0.7 nicht mehr möglich ist, explizit eine Aspektinstanz zur Laufzeit zu erzeugen und über `addObject` dieser Instanz ein Objekt zuzuordnen (vgl. [RS03]), lassen sich die oben beschriebenen Realisierungsansätze mit ASPECTJ nicht mehr umsetzen.

4.7.4 DARWIN und LAVA

Das DARWIN-Modell

In der ersten Version des DARWIN-Modells ([Kni96]) werden allgemeine Anforderungen an eine objektbasierte Vererbung spezifiziert.[50] Diese dient dann als Grundlage für ein neues Rollenmodell. Für die objektbasierte Vererbung existieren die beiden Konzepte **Delegation** (*Delegation*) und **Konsultation** (*Consultation*). Der Unterschied dieser beiden Konzepte wird in der Folge an einem JAVA-Beispiel dargestellt.[51]

Gegeben seien die beiden Klassen aus Prg. 1 und Prg. 2 (S. 83).

```
 1:  public class SuperClass {
 2:
 3:    public void mA() {
 4:      System.out.println ( "mA() aus SuperClass" );
 5:      this.mB();
 6:      this.mC();
 7:    } // mA
 8:
 9:    void mB() {
10:      System.out.println ( "  mB() aus SuperClass" );
11:    } // mB
12:
13:    void mC() {
14:      System.out.println ( "  mC() aus SuperClass" );
15:    } // mB
16:
17:  } // SuperClass
```

Prg. 1 Die Klasse `SuperClass`

Die Ausführung der `main`-Methode von `SubClass` liefert bei der Verwendung der Delegation das folgende Ergebnis:

```
mD() aus SubClass
mA() aus SuperClass
  mB() aus SubClass
  mC() aus SuperClass
```

Innerhalb des Aufrufs der Oberklassenmethode `mA` wird also die Methode `mB` aus der Unterklasse verwendet. Die Referenz `this` aus Zeile `5` von Prg. 1 wird bei der Delegation an das aufrufende Objekt gebunden, weshalb die redefinierte Methode aus `SubClass` benutzt wird.

[50] Das DARWIN-Modell wurde innerhalb des DARWIN-Projekts entwickelt. Weitere Informationen zu dem Projekt finden sich unter `http://javalab.cs.uni-bonn.de/research/darwin/index.html` ([The03b]).

[51] JAVA unterstützt natürlich nur klassenbasierte Vererbung und Delegation.

```
 1: public class SubClass extends SuperClass {
 2:
 3:   public void mD() {
 4:     System.out.println ( "mD() aus SubClass" );
 5:     this.mA();
 6:   } // mD
 7:
 8:   void mB() {
 9:     System.out.println ( "  mB() aus SubClass" );
10:   } // mB
11:
12:   public static void main ( String[] args ) {
13:     SubClass sc = new SubClass();
14:     sc.mD();
15:   } // main
16:
17: } // SubClass
```

Prg. 2 Die Klasse `SubClass`

Das Konzept der Konsultation würde dagegen folgende Ausgabe erzeugen:

```
mD() aus SubClass
mA() aus SuperClass
  mB() aus SuperClass
  mC() aus SuperClass
```

Hier wird die Referenz `this` an das Oberklassenobjekt gebunden, was zum Aufruf der Methode `mB` aus der Oberklasse führt.

Das DARWIN-Modell bietet für die objektbasierte Vererbung sowohl Delegation als auch Konsultation an. In einer Delegations- bzw. Konsultationsbeziehung wird die Oberklasse als *Parent*-Klasse und die Unterklasse als *Child*-Klasse bezeichnet. Zwischen einer Child-Klasse und einer Parent-Klasse besteht eine *Subtype*-Beziehung, d.h. eine Child-Klasse besitzt an ihrer Schnittstelle alle Methoden der Parent-Klasse.

Abb. 36 (S. 84) zeigt die wesentlichen Eigenschaften des DARWIN-Modells:

- Auf der Klassenebene ist Mehrfachvererbung erlaubt (z.B. besitzt die Klasse `A2` die beiden direkten Oberklassen `A` und `B`).
- Auf der Objektebene ist Mehrfachkonsultation und Mehrfachdelegation möglich. Beispielsweise besitzt die Child-Klasse `C` die Parent-Klassen `A` und `D`, für die jeweils eine Delegationsbeziehung modelliert ist. Eine Child-Klasse darf natürlich auch gleichzeitig Delegations- und Konsultationsbeziehungen zu Parent-Klassen haben.
- Eine Parent-Klasse kann selbst wieder die Rolle einer Child-Klasse spielen. Damit kann eine Child-Klasse sowohl direkte als auch indirekte Parent-Klassen besitzen. Die Child-Klasse `F` hat z.B. die direkte Parent-Klasse `E` und die indirekte Parent-Klasse `B`; `B` ist die direkte Parent-Klasse der Child-Klasse `E`.

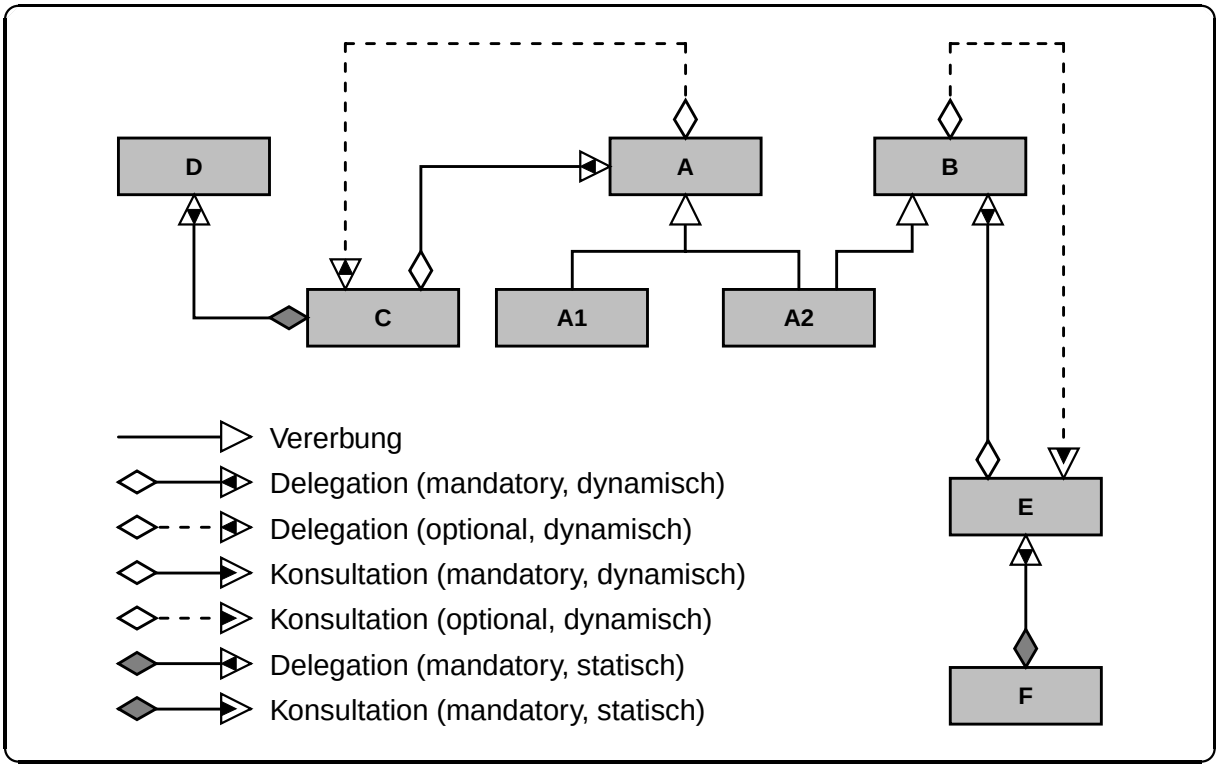


Abb. 36 Das DARWIN-Modell

- Ein Attribut einer Child-Klasse, über das eine Konsultations- oder Delegationsbeziehung zu einer Parent-Klasse hergestellt wird, kann entweder *mandatory* oder *optional* sein. Ein *mandatory*-Attribut muß immer einen konkreten Wert besitzen. Die Delegations- bzw. Konsultationsbeziehung wird entsprechend als *mandatory* oder *optional* bezeichnet. Liegt ein *optional*-Attribut vor, so ist bei jeder Nachricht, die von dem Child-Objekt nicht bearbeitet werden kann, und deshalb an das Parent-Objekt weitergeleitet werden muß, zur Laufzeit zu überprüfen, ob das Parent-Objekt gerade existiert. Die Klasse `C` besitzt ein *mandatory*-Attribut für die Delegationsbeziehung zu `A`, während zwischen den Klassen `B` und `E` eine optionale Konsultationsbeziehung besteht.
- Eine zyklische Delegation bzw. Konsultation ist erlaubt. Allerdings muß in einem Zyklus mindestens eine optionale Beziehung enthalten sein. Die Klassen `B` und `E` sind in einem Zyklus enthalten.
- Das Attribut zur Realisierung einer Delegations- oder Konsultationsbeziehung kann statisch oder dynamisch sein. Ist das Attribut statisch, darf sich sein Wert nach der Initialisierung im Konstruktor nicht mehr ändern. Ist das Attribut dagegen dynamisch, so kann das Parent-Objekt zur Laufzeit ausgetauscht werden. Insbesondere ist es damit zur Laufzeit möglich, Verhaltensänderungen durchzuführen, da dem Attribut auch eine Referenz auf ein Unterklassenobjekt zugewiesen werden darf. Ein Beispiel ist die Delegationsbeziehung zwischen den Klassen `C` und `A`. Dem Attribut in einem `C`-Objekt dürfen auch Referenzen auf Objekte der Klassen `A1` und `A2` zugewiesen werden. Wenn jetzt `A1` eine Methode aus `A` redefiniert, ändert sich beim Wechsel von einem `A`-Objekt zu einem `A1`-Objekt das Laufzeitverhalten beim Aufruf dieser Methode.

Das Rollenmodell

Für das Rollenmodell wird in [Kni96] die folgende Struktur vorgeschlagen:

- **Konzeptuelles Objekt** (*Conceptual Object*): Ein konzeptuelles Objekt besteht aus einer essentiellen Rolle und einer eventuell leeren Menge von transienten Rollen.
- **Essentielle Rolle** (*Essential Role*): Diese Rolle existiert während der gesamten Lebensdauer des konzeptuellen Objekts und stellt den Ausgangspunkt für die Rollenhierarchie dar.
- **Transiente Rolle** (*Transient Role*): Eine transiente Rolle kann dynamisch angenommen und wieder aufgegeben werden.
- Ein konzeptuelles Objekt kann zu einem Zeitpunkt mehrere transiente Rollen besitzen.[52]
- Eine transiente Rolle kann selbst wieder transiente Unterrollen besitzen.
- Das konzeptuelle Objekt besitzt eine eigenständige Identität.

Ein wesentlicher Unterschied zu anderen Rollenmodellen ist die Interpretation des Identitätsbegriffs. Über jede Rolle kann auf **alle** Methoden des konzeptuellen Objekts zugegriffen werden. Für das Beispiel aus Abb. 37 gilt daher, daß die Frau *Sally* in ihrer Rolle als Patientin auch nach

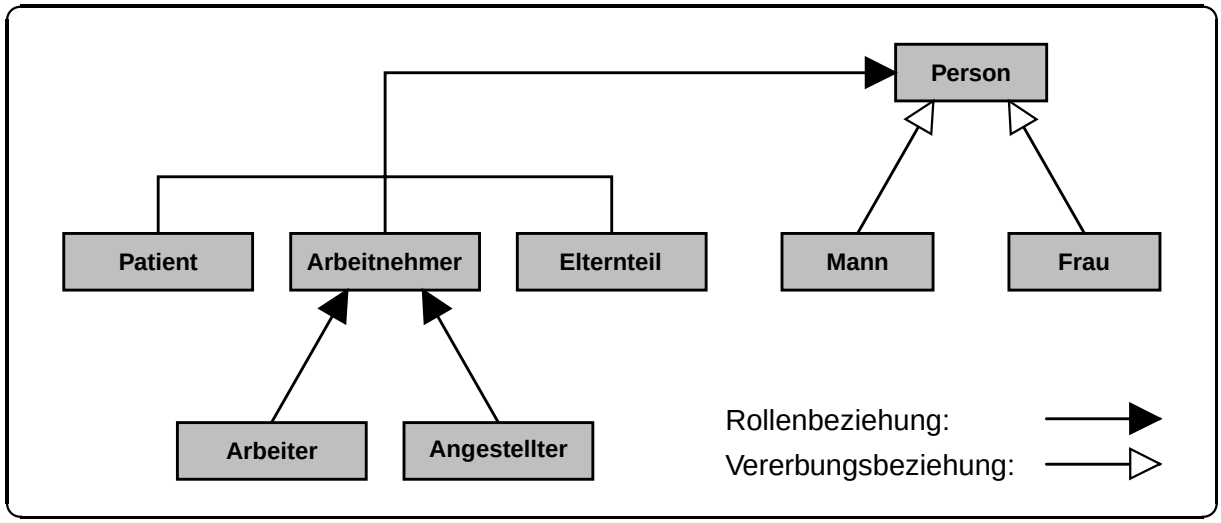


Abb. 37 Das Rollenmodell einer Person

ihrer Tochter gefragt werden kann, was einen Zugriff auf die Elternteilrolle bedeutet. Wenn in einer Rolle die Methode zu einer Nachricht nicht vorhanden ist, gelten die folgenden Regeln:

- Eine Rolle darf das Verhalten ihrer direkten und indirekten Oberrollen verändern, d.h. die Nachricht wird gemäß dem Delegationskonzept an die Oberrolle weitergeleitet.
- Für alle anderen Rollen wird das Konsultationsprinzip verwendet.

Bei diesem Modellierungsansatz für ein Rollenmodell können natürlich Namenskonflikte auftreten. Wenn beispielsweise *Sally* in ihrer Rolle als Frau nach einer Information gefragt wird, für die sowohl eine Methode in ihrer Patientenrolle als auch in ihrer Arbeitnehmerrolle vorliegt, so wird in [Kni96] nicht näher spezifiziert, welche Methode letztlich ausgeführt wird. Die zu-

[52] Auf die Frage, ob eine Rolle mehrere direkte Oberrollen haben darf, wird in [Kni96] nicht näher eingegangen.

gehörige Aussage lautet: *„In our opinion the requesting entity might prefer a general answer over no answer, and no answer (resp. being forced to state the request more precisely) over a useless one“*. Als eine Möglichkeit zur Auflösung eines derartigen Namenskonflikts wird die Umbenennung von Methoden vorgeschlagen.

Das Rollenmodell läßt sich sehr einfach in ein DARWIN-Modell abbilden:

- Zwischen einer Unterrolle und ihrer direkten Oberrolle wird eine statische mandatory-Delegationsbeziehung modelliert.
- Zwischen einer Oberrolle und einer direkten Unterrolle besteht eine optionale, dynamische Konsultationsbeziehung.

Die Abbildung des Rollenmodells aus Abb. 37 (S. 85) in ein DARWIN-Modell ist in Abb. 38 dargestellt.

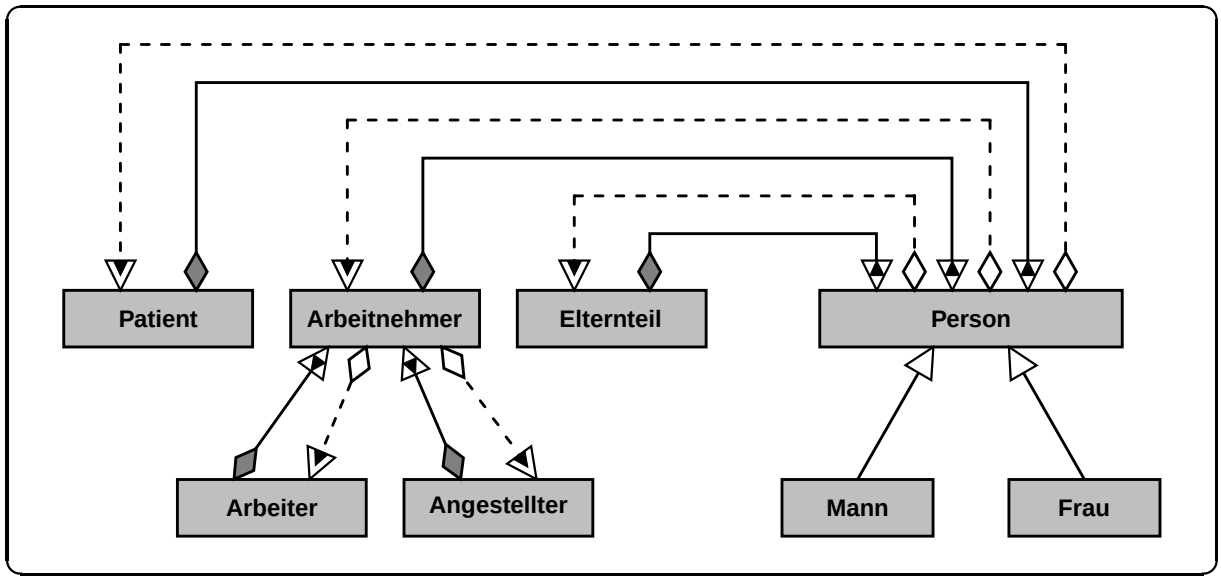


Abb. 38 Das DARWIN-Modell für das Rollenmodell aus Abb. 37 (S. 85)

Die Umsetzung des Rollenmodells in LAVA und JAVA

In [Sie98] erfolgt die Implementierung des Rollenmodells in den Programmiersprachen LAVA und JAVA. LAVA ([CKC99]) baut auf JAVA auf und ergänzt die Sprache um objektbasierte Vererbung. Sowohl Mehrfachdelegation als auch Mehrfachkonsultation werden unterstützt.

Zur Spezifikation eines Rollenmodells wurde zunächst die JAVA-Klassendeklaration um die `isRoleClassOf`-Klausel erweitert. Die Deklaration der Rollenbeziehung zwischen `Person` und `Arbeitnehmer` hat damit die folgende Form:

```
class Arbeitnehmer isRoleClassOf Person {
  ...
}
```

Für die Umsetzung nach LAVA und JAVA wurden zwei Prä-Compiler geschrieben. Diese erzeugen für jede Rollenklasse `X` die folgenden Methoden, die in der direkten Oberrolle zur Verfü-

gung stehen:

- `becomeX(`*`Parameterdeklaration`*`)`: Ein Objekt übernimmt die Rolle der Rollenklasse `X`.
- `abandonX()`: Ein Objekt gibt die Rolle der Rollenklasse `X` auf.
- `asX()`: Eine Referenz auf das Rollenobjekt der Rollenklasse `X` wird als Ergebnis zurückgeliefert.
- `isX()`: Es wird getestet, ob die Rolle der Rollenklasse `X` bereits existiert.

Die weitere Umsetzung des Rollenmodells in die LAVA-Sprache war vergleichsweise einfach: Für jede Rolle wurde eine eigene Klasse erstellt. Die Realisierung der Rollenbeziehungen erfolgte unter Verwendung entsprechender Delegations- und Konsultationsattribute.[53]

Da JAVA keine objektbasierte Vererbung kennt, wurden hier die Delegations- und Konsultationsmechanismen simuliert. Um für jede Rollenklasse (einschließlich der essentiellen Rolle) die Funktionalität des konzeptuellen Objekts zugreifbar zu machen, wurde ein JAVA Interface generiert, das alle Methoden der gesamten Rollenhierarchie enthält. Jede Klasse der Rollenhierarchie implementiert dann dieses Interface. Existiert eine Rolle noch nicht, so führt der Aufruf einer ihrer Methoden zu einer Exception. Eine ausführliche Beschreibung der JAVA-Implementierung ist in [Sie98] enthalten.

Die aktuellen DARWIN- und LAVA-Versionen

Die aktuellen Versionen des DARWIN-Modells und der Sprache LAVA sind in [Kni00] beschrieben. Die Funktionalität des DARWIN-Modells wurde deutlich eingeschränkt:

- Die Delegations- und Konsultationsbeziehungen müssen die *mandatory*-Eigenschaft besitzen. Als optional gekennzeichnete Delegations- und Konsultationsattribute sind nicht mehr erlaubt ([Kni00, S. 66]).
- Mehrfachdelegation- und Mehrfachkonsultation werden nicht mehr betrachtet (vgl. Fußnote 12 in [Kni00, S. 73]).

Die unmittelbare Folge aus der ersten Einschränkung ist, daß zyklische Delegations- und Konsultationsbeziehungen nicht mehr möglich sind.[54] Damit kann aber auch das Konzept einer Identität für das konzeptuelle Objekt nicht aufrecht erhalten werden.[55] Die zweite Version der LAVA-Sprache wurde an diese Einschränkungen angepaßt. Konsequenterweise ist die oben beschriebene Umsetzung des Rollenmodells jetzt nicht mehr möglich.[56]

[53] Dieses sind Attribute, die mit den LAVA-spezifischen Modifikatoren `delegatee` bzw. `consultee` versehen sind und die Delegations- bzw. Konsultationsbeziehung zwischen zwei Klassen herstellen.

[54] Dafür war mindestens eine optionale Beziehung notwendig.

[55] Hierfür war gerade ein Zyklus aus einer Delegations- und einer Konsultationsbeziehung erforderlich.

[56] Die aktuelle Version des LAVA Compilers (lavac 0.21, siehe [The03c]) liefert bei dem Versuch, eine zyklische Rollenbeziehung umzusetzen, neben einer `cyclic inheritance`-Fehlermeldung zusätzlich eine interne `java.lang.StackOverflowError` Exception.

Bewertung des Rollenmodells

Das Rollenmodell besitzt die folgenden Nachteile:

- Eine Rollenkardinalität größer als 1 ist nicht möglich.
- Restriktionen für Rollen sind nicht vorgesehen.

Durch die Neudefinition des DARWIN-Modells ist die als wesentlich herausgestellte Eigenschaft der Identität des konzeptuellen Objekts verloren gegangen.

4.8 Vergleich der Rollenmodelle und ihrer Realisierungen

Eine gegenüberstellende Zusammenfassung der wichtigsten Eigenschaften der vorgestellten Rollenmodelle und Realisierungen ist in Abb. 39 (S. 91) bis Abb. 41 (S. 93) enthalten. Dabei werden für ein Rollenmodell die Modelleigenschaften und die Realisierungseigenschaften dargestellt. Falls für einen Ansatz keine Realisierung beschrieben oder die Realisierung nur skizziert wurde, sind die zugehörigen Felder der Tabellenzeile grau hinterlegt. Weiße, nicht ausgefüllte Felder bedeuten, daß zu der entsprechenden Eigenschaft keine Informationen vorliegen. Bei den Ansätzen, die auf Mustern basieren, wurden keine Modelleigenschaften angegeben, da diese mit den Realisierungseigenschaften identisch sind. Diesselbe Aussage gilt für die ersten drei Ansätze aus dem Programmiersprachenbereich.[57] Bei den Ansätzen aus dem Musterbereich wurde JAVA als Implementierungssprache angenommen.

Die verwendeten Tabelleneinträge und Abkürzungen haben die folgende Bedeutung:

- M: Modell
- R: Realisierung
- DB-Bereich: Datenbankbereich
- PS-Bereich: Programmiersprachenbereich
- Struktur
  - B(1): Es liegt eine Baumstruktur mit einer Wurzelrolle und einer Unterrollenebene vor. Nur die Wurzelrolle darf Unterrollen besitzen.
  - B(k): Es liegt eine Baumstruktur mit beliebig vielen Ebenen vor. Daher können auch Unterrollen wieder Unterrollen besitzen.
  - G(1): Die Rollenstruktur ist ein azyklischer Graph mit einer Wurzelrolle. Für Rollen ist damit Mehrfachvererbung möglich.
  - G(n): Die Rollenstruktur kann zusätzlich zu G(1) mehrere Wurzelrollen besitzen.
- Mehrfachrollen
  - M(1,1): Das Objekt (d.h. die Wurzelrolleninstanz) kann zu einem Zeitpunkt nur eine Unterrolleninstanz besitzen.
  - M(n,1): Das Objekt kann simultan mehrere Unterrolleninstanzen besitzen, von jedem Rollentyp aber höchstens eine Instanz.
  - M(n,m): Das Objekt kann beliebig viele Unterrolleninstanzen unterschiedlicher Unterrollentypen besitzen.
- Dynamik
  - ja: Das Objekt kann zur Laufzeit neue Rollen übernehmen.
  - nein: Ein Objekt wird immer mit allen seinen Rolleninstanzen erzeugt.
- Identität
  - ZR: Eine Rolle schränkt die Objektsicht ein, d.h. über eine Rollenreferenz ist nur der Zugriff auf die Methoden der Rolle selbst und auf die Methoden aller direkten und indirekten Oberrollen möglich.

[57] Bei den entsprechenden Arbeiten hat der Modellaspekt eine eher untergeordnete oder gar keine Rolle gespielt.

  - ZG: Über eine Rollenreferenz kann auf alle Methoden des Gesamtobjekts zugegriffen werden.
  - E: Ein Erzeugen einer Rolleninstanz ist nur im Kontext des Gesamtobjekts möglich. Mit Ausnahme einer Wurzelrolleninstanz ist die Erzeugung aller Rolleninstanzen von der Existenz (mindestens) einer direkten Oberrolleninstanz abhängig. Somit ist auch die Eigenschaft der Abhängigkeit einer Rolleninstanz von ihren Oberrolleninstanzen gewährleistet.
  - L: Das Löschen einer Rolleninstanz `r` ist nur im Rahmen des Gesamtobjekts möglich. Insbesondere werden alle Unterrolleninstanzen von `r` ebenfalls gelöscht.
  - nein: Es liegt keine eigenständige Gesamtobjektidentität vor. Stattdessen wird eine Menge von unabhängigen Objekten zur Realisierung des Rollenmodells verwendet.
- Redefinition
  - V: Eine Rolle kann eine Methode einer Oberrolle nur vollständig redefinieren. Dieses ist vergleichbar mit dem Konzept der Konsultation (siehe Abschnitt 4.7.4 (S. 82)).
  - P: Eine Oberrollenmethode `m` kann partiell redefiniert werden, indem die Unterolle eine Methode `m1` redefiniert, die von `m` aufgerufen wird. Dieses ist mit dem Konzept der Delegation vergleichbar (siehe Abschnitt 4.7.4 (S. 82)).
- Restriktionen
  - ja: Restriktionen werden auf der Modell- bzw. Realisierungsebene unterstützt.
  - nein: Restriktionen werden auf der entsprechenden Ebene nicht unterstützt.
- Migration
  - ja: Eine Rollenmigration ist möglich.
  - nein: Eine Rollenmigration ist nicht möglich.
- Typsicherheit
  - ja: Es wird eine Programmiersprache mit statischer Typprüfung verwendet. Ein „ja“ wird auch vergeben, wenn zur Laufzeit mit Typkonvertierungen gearbeitet werden muß (z.B. durch Anwendung des Cast-Operators in JAVA), obwohl in diesem Fall eine 100%-ige Typsicherheit nicht mehr gegeben ist.
  - nein: Es wird eine Sprache ohne statische Typprüfung verwendet.

Weitere Vergleiche von Rollenmodellen sind in [Ste00], [Kni96] und [KRS98] enthalten, die aber stärker die Modellierungs- als die Realisierungseigenschaften in den Vordergrund stellen.

	Struktur		Mehrfach-rollen		Dynamik		Identität		Redefini-tion		Restrik-tionen		Migration		Typsicher-heit	
	M	R	M	R	M	R	M	R	M	R	M	R	M	R	M	R
Object-Role-Model 4.2.2 (S. 23) DB-Bereich	B(1)	B(1)	M(n,1)	M(n,1)	ja	ja	ZG	ZG	nein	nein	ja	ja	nein	nein		
FIBONACCI 4.2.3 (S. 24) DB-Bereich	G(1)	G(1)	M(n,1)	M(n,1)	ja	ja	ZR,E	ZR,E	V,P	V,P	nein	nein	nein	nein		ja
DOOR 4.2.4 (S. 25) DB-Bereich	B(k)	B(k)	M(n,m)	M(n,m)	ja	ja	ZG	ZG			ja	ja	ja	ja		
ORM 4.3.1 (S. 26) Analyse	B(1)		M(n,m)		ja		ZR, E,L				ja					
KRISTENSEN und ØSTERBYE 4.3.2 (S. 27) Analyse	B(k)		M(n,m)		ja		ZR, ZG		V		ja		ja			
OORAM 4.4.1 (S. 42) Kollaboration	B(1)		M(n,m)		nein		ZG, E,L		V		nein		nein			
VAN HILST und NOTKIN 4.4.2 (S. 45) Kollaboration	B(1)	B(1)	M(n,1)	M(n,1)	nein	nein	ZG, E,L	ZG, E,L	V,P	V,P	nein	nein	nein	nein		ja
Framework-Entwurf 4.4.3 (S. 46) Kollaboration	B(1)		M(n,1)		ja		ZG, E,L		V,P		ja		nein			
EPSILON 4.4.4 (S. 49) Kollaboration	B(1)	B(1)	M(n,m)	M(n,m)	ja	ja	ZR	ZR	V,P	V,P	nein	nein	ja	ja		ja
KENDALL 4.5.2 (S. 51) Agenten	B(1)		M(n,1)		ja		ZR		V,P		nein		nein			

Abb. 39 Vergleich der Rollenmodelle – Teil 1

	Struktur		Mehrfach-rollen		Dynamik		Identität		Redefini-tion		Restrik-tionen		Migration		Typsicher-heit	
	M	R	M	R	M	R	M	R	M	R	M	R	M	R	M	R
XROLE 4.5.3 (S. 52) Agenten	B(1)		M(n,1)		ja		ZR				ja		nein			
ROLEEP 4.5.4 (S. 53) Agenten	B(1)	B(1)	M(n,1)	M(n,1)	ja	ja	ZR	ZR	V,P	V,P	nein	nein	ja	ja		ja
Single-Role-Type 4.6.2 (S. 56) Muster		B(1)		M(1,1)		ja		ZG		nein		nein		nein		ja
Separate-Role-Type 4.6.3 (S. 57) Muster		G(n)		M(n,m)		ja		nein		V,P		nein		nein		ja
Subtype 4.6.4 (S. 57) Muster		B(k)		M(n,m)		ja		nein		V,P		nein		nein		ja
Internal-Flag 4.6.5 (S. 61) Muster		B(1)		M(n,1)		ja		ZG,E		V,P		ja		nein		ja
Hidden-Delegate 4.6.6 (S. 63) Muster		B(1)		M(n,1)		ja		ZG,E		V,P		nein		nein		ja
State-Object 4.6.7 (S. 64) Muster		B(1)		M(1,1)		ja		ZG,E		V,P		nein		nein		ja
Role-Object 4.6.8 (S. 66) Muster		B(k)		M(n,m)		ja		ZR, E,L		V		nein		nein		ja
Extended-SMALLTALK 4.7.1 (S. 69) PS-Bereich		B(k)		M(n,m)		ja		ZR, E,L		V,P		nein		nein		nein

Abb. 40 Vergleich der Rollenmodelle – Teil 2

	Struktur		Mehrfach-rollen		Dynamik		Identität		Redefini-tion		Restrik-tionen		Migration		Typsicher-heit	
	M	R	M	R	M	R	M	R	M	R	M	R	M	R	M	R
JAVA Role API 4.7.2 (S. 71) PS-Bereich		B(k)		M(n,m)		ja		ZR, E		nein		nein		nein		ja
ASPECTJ 0.2 (Hybrid-Ansatz) 4.7.3 (S. 75) PS-Bereich		B(1)		M(n,m)		ja		ZG, E,L		V,P		nein		nein		ja
DARWIN und LAVA (Version 1) 4.7.4 (S. 82) PS-Bereich	B(k)	B(k)	M(n,1)	M(n,1)	ja	ja	ZR,E ZG,L	ZR,E ZG,L	V,P	V,P	nein	nein	nein	nein	ja	ja
DARWIN und LAVA (Version 2) 4.7.4 (S. 87) PS-Bereich	B(k)	B(k)	M(n,1)	M(n,1)	ja	ja	ZR, E,L	ZR, E,L	V,P	V,P	nein	nein	nein	nein	ja	ja

Abb. 41 Vergleich der Rollenmodelle – Teil 3

4.9 Kapitelzusammenfassung

In diesem Kapitel wurden alternative Ansätze für die Modellierung und Realisierung von Rollenkonzepten vorgestellt. Diese Ansätze können in zwei große Klassen untergliedert werden: Rollenkonzepte zur Modellierung des dynamischen Vererbungscharakters und Rollenkonzepte für die Modellierung von Objektkollaborationen. Innerhalb der ersten Klasse wurden Konzepte aus dem Datenbankumfeld, aus der Analysephase, aus dem Analyse- und Entwurfsmusterbereich und dem Programmiersprachenbereich betrachtet. Für die zweite Klasse wurden Konzepte zur Modellierung der Objektzusammenarbeit und aus dem Umfeld der Agentensysteme vorgestellt.

Am Ende des Kapitels wurde eine Bewertung dieser Modelle anhand von acht Kriterien vorgenommen:

- Welche Struktur besitzt das Modell (Baum oder Graph)?
- Sind Mehrfachrollen erlaubt?
- Können Rollen dynamisch zur Laufzeit übernommen und wieder abgegeben werden?
- Wird für das Gesamtobjekt, d.h. das Objekt mit allen seinen aktuellen Rollen, ein Identitätsbegriff unterstützt?
- In welcher Form ist eine Redefinition von Methoden möglich?
- Lassen sich für das Rollenmodell Restriktionen spezifizieren?
- Wird eine Rollenmigration unterstützt?
- Liegt Typsicherheit vor?

Die Kriterien wurden jeweils für das Konzept und die Realisierung getrennt betrachtet. Dabei stellte sich heraus, daß keines der Modelle gleichzeitig die folgenden, wünschenswerten Eigenschaften besitzt:

- Als Struktur wird ein azyklischer Graph unterstützt.
- Von einem Rollentyp können mehrere Rollenobjekte existieren.
- Es lassen sich auf der Modellierungsebene Restriktionen spezifizieren, die automatisch in eine Implementierung integriert werden.
- Es liegt eine vollständige Typsicherheit vor. Vollständige Typsicherheit soll in diesem Zusammenhang bedeuten, daß bei der Verwendung einer Implementierungssprache mit statischer Typprüfung für die Anwendung des Rollenmodells innerhalb einer Applikation keine Typkonvertierungen notwendig sind, die dann möglicherweise zu Laufzeitfehlern führen.

Im nächsten Kapitel wird ein neues Rollenkonzept vorgestellt, das diese Eigenschaften erfüllt.

Kapitel 5

Ein neues Konzept zur Realisierung von Rollenmodellen mit Mehrfachvererbung

5.1 Einleitung

Die Grundidee des neuen Ansatzes besteht in der Trennung der Strukturbeschreibung eines Rollenmodells von der Funktionalitätsbeschreibung der einzelnen Rollen. Für die Spezifikation der Rollenmodellstruktur wird eine Beschreibungssprache definiert. Die Funktionalität einer Rolle wird über ihre ***korrespondierende Klasse*** beschrieben, wobei JAVA als Programmiersprache dient. Im einfachsten Fall bestehen zwischen den korrespondierenden Klassen weder Assoziations- noch Vererbungsbeziehungen. Die zur Realisierung des Rollenmodells notwendigen Beziehungen werden in den generierten Rollenklassen hergestellt. Zu jeder korrespondierenden Klasse wird eine Rollenklasse erzeugt, welche die Zugriffe auf die korrespondierende Klasse kapselt. Innerhalb des Generierungsprozesses werden dann auch alle Tests erzeugt, die für eine Umsetzung der Restriktionen und der Konsistenzbedingungen für Rollenmodelle mit Mehrfachvererbung notwendig sind.

Für die Konzeption und Entwicklung eines neuen Rollenmodells wird in den Abschnitten 5.2 (S. 97) bis 5.7 (S. 181) ein inkrementeller Ansatz verwendet:

- Abschnitt 5.2 (S. 97) beschreibt die Analyse, den Entwurf und die Generierung einer Rollenhierarchie mit Einfachvererbung.

  Zunächst wird von dem einfachsten Fall, einer vollständigen Unabhängigkeit der korrespondierenden Klassen, ausgegangen. Danach werden die Modelländerungen vorgestellt, die durch die Hinzunahme von Assoziations- und Vererbungsbeziehungen zwischen den korrespondierenden Klassen entstehen.
- In Abschnitt 5.3 (S. 139) werden Unterrollenkardinalitäten mit Werten größer als 1 integriert.

  Beim Erzeugen von Unterrollen ist jetzt der Test zu modifizieren, ob bereits alle Unterrollen existieren. Außerdem ändert sich die Operation zum Löschen von Unterrollen.
- Die Mehrfachvererbung für Rollenmodelle betrachtet Abschnitt 5.4 (S. 149).

Es werden Rollenmodelle mit mehreren Wurzelrollen unterstützt, deren Graphstruktur zyklenfrei ist und keine transitiven Kanten enthält. Durch die neue Eigenschaft der Mehrfachvererbung werden die Tests, die beim Erzeugen einer Unterolle auszuführen sind, wesentlich komplexer. Außerdem tritt das Problem auf, daß eine Methode in mehreren Oberrollen definiert sein kann. Als weiterer Punkt ist zu beachten, daß es jetzt möglich ist, inkonsistente Kardinalitäten für Unterrollen zu spezifizieren.

- Wie sich in dem gewählten Ansatz das Polymorphismuskonzept unterstützen läßt, behandelt Abschnitt 5.5 (S. 171).

  Da die generierten Rollenklassen nicht über das Vererbungskonzept miteinander verbunden sind, ist das Polymorphismuskonzept hier zunächst nicht anwendbar. Durch die Generierung und Implementierung entsprechender Schnittstellen wird dieses Problem behoben.

- Der Abschnitt 5.6 (S. 174) befaßt sich mit dem Thema Restriktionen.

  Es werden die Restriktionstypen **and**, **or**, **exor** und **nocreation** betrachtet. Für die beiden letzten Restriktionen erfolgt die Formulierung von Bedingungen, die einzuhalten sind, um die Konsistenz des Rollenmodells nicht zu verletzen.

- Das Konzept der abstrakten Rollen wird in Abschnitt 5.7 (S. 181) vorgestellt.

  Hier wird die Frage behandelt, wie sich das Konzept der abstrakten Klassen auf Rollen übertragen läßt.

Auf weitere, potentielle Ergänzungen der Funktionalität des Rollenmodells wird in Abschnitt 5.8 (S. 190) eingegangen, bevor in Abschnitt 5.9 (S. 196) die Integration des Rollenmodells in den Software-Entwicklungsprozeß besprochen wird. Den inhaltlichen Abschluß des Kapitels stellt in Abschnitt 5.10 (S. 197) die Beschreibung der implementierten Software-Architektur zur Generierung von Rollenmodellen dar.

5.2 Verwendung einer Rollenhierarchie mit Einfachvererbung

5.2.1 Analysemodell

Für die Beschreibung der einzelnen Rollen, die ein Objekt innerhalb einer Rollenhierarchie annehmen kann, liegen die Klassen $\mathbf{Class}_i$ (i=1,. . .,n) vor (vgl. Abb. 42). Über die Klasse $\mathbf{Class}_i$ wird festgelegt, welche zusätzlichen Attribute und Operationen ein Objekt erhält, wenn es diese Rolle übernimmt. Um die Struktur des entstehenden Rollenmodells einfach zu halten, sollen für die Klassen $\mathbf{Class}_1$ bis $\mathbf{Class}_n$ zunächst die folgenden Restriktionen gelten:

- Zwischen den Klassen bestehen keine Assoziationen.
- Die Klassen haben keine Oberklassen[1].

Diese beiden Aspekte werden in den Abschnitten 5.2.5 (S. 119) und 5.2.6 (S. 126) behandelt.

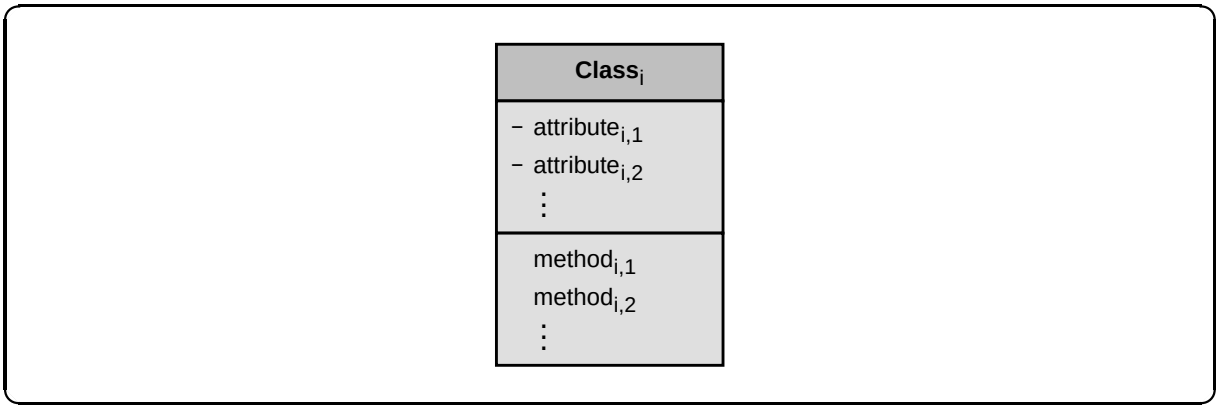


Abb. 42 Klasse zur Beschreibung einer Rolle in UML-Notation ([HK99])

Die Klassen $\mathbf{Class}_1$ bis $\mathbf{Class}_n$ werden zur Definition der Rollenhierarchie in einer ***Baumstruktur*** angeordnet, um die Einfachvererbung zu realisieren. Dabei stellt $\mathbf{Class}_1$ die Wurzel des Baumes dar. Die verbleibenden Klassen $\mathbf{Class}_2$ bis $\mathbf{Class}_n$ sind jeweils genau einmal im Baum enthalten. Abb. 43 (S. 98) zeigt ein Beispiel mit 8 Klassen. Für die Rollenbeziehung wird die in Abb. 9 (S. 16) eingeführte Notation verwendet. Um den dynamischen Charakter der Rollenerzeugung darzustellen, enthält das Diagramm zusätzlich die erlaubten Kardinalitäten:

- Eine Unterrolle ***muß*** genau eine direkte Oberrolle besitzen.[2]
- Die Oberrolle ***kann*** von jedem erlaubten, direkten Unterrollentyp jeweils maximal eine Unterrolle besitzen.

Die Klassen $\mathbf{Class}_1$ bis $\mathbf{Class}_n$ dienen ausschließlich zur Festlegung der Funktionalität der einzelnen Rollen. Zusätzlich werden aber Operationen zur ***Verwaltung des Rollenmodells*** eines Objekts benötigt:

- ***Erzeugung*** von Unterrollen:
  Jede Rolle besitzt Erzeugeoperationen für alle direkten Unterrollen. Durch die Einschrän-

[1] Eine Ausnahme stellen natürlich die in der verwendeten Programmiersprache standardmäßig vorhandenen Oberklassen dar. Im Falle von JAVA ist dies die Klasse **Object**.

[2] Die Kardinalität 1 wird bei der Rollenbeziehung nicht in das Klassendiagramm eingetragen.

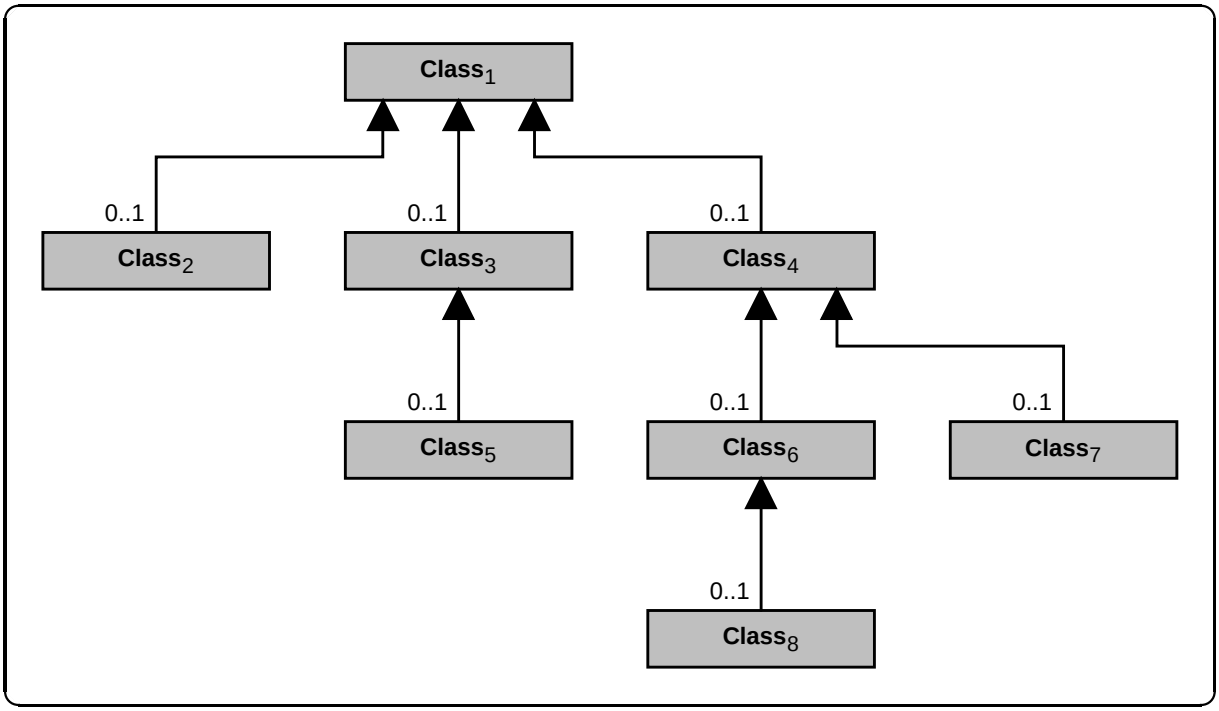


Abb. 43 Definition der Rollenhierarchie über eine Baumstruktur

kung auf die direkten Unterrollen wird sichergestellt, daß nur konsistente Rollen erzeugt werden; beispielsweise ist es nicht sinnvoll, einer Person die Rolle eines wissenschaftlichen Mitarbeiters zuzuordnen, wenn diese Person nicht bereits die Rolle UniMitarbeiter innehat.

- ***Löschen*** einer Rolle:
  Wird eine Rolle gelöscht, werden auch alle vorhandenen Unterrollen gelöscht, um die Konsistenz des Rollenmodells für das betroffene Objekt zu erhalten; es darf nicht passieren, daß die Rolle eines UniMitarbeiters gelöscht wird, seine Rolle als wissenschaftlicher Mitarbeiter jedoch bestehen bleibt. Als Alternative wäre es natürlich denkbar, das Löschen einer Rolle zu verbieten, solange noch Unterrollen existieren.
- ***Existenztest*** für eine Rolle:
  Da bei der vorliegenden Rollenmodellvariante eine Rolle höchstens einmal angenommen werden kann, ist es sinnvoll abfragen zu können, ob eine bestimmte Rolle bereits existiert.

5.2.2 Entwurfsmodell

Für die Umsetzung eines Rollenmodells werden zu den Klassen **Class**$_1$ bis **Class**$_n$ eigene Rollenklassen **RoleClass**$_1$ bis **RoleClass**$_n$ entworfen. **Class**$_i$ ist dabei die zu **RoleClass**$_i$ ***korrespondierende*** Klasse, die wesentlich die Struktur der Rollenklasse definiert. Eine Klasse **RoleClass**$_i$ hat die zentrale Aufgabe, den Vererbungscharakter einer Rolle zu simulieren. Daneben besitzt sie die zur Verwaltung des Rollenmodells erforderlichen Operationen. Ein entscheidender Punkt zur Erfüllung dieser beiden Anforderungen ist der Aufbau und die Verwaltung geeigneter Beziehungen zwischen den beteiligten Objekten der Klassen **Class**$_1$ bis **Class**$_n$ und **RoleClass**$_1$ bis **RoleClass**$_n$. Abb. 44 (S. 99) zeigt das Klassendiagramm, welches entsteht, wenn das in Abb. 43 definierte Rollenmodell zugrunde gelegt wird.

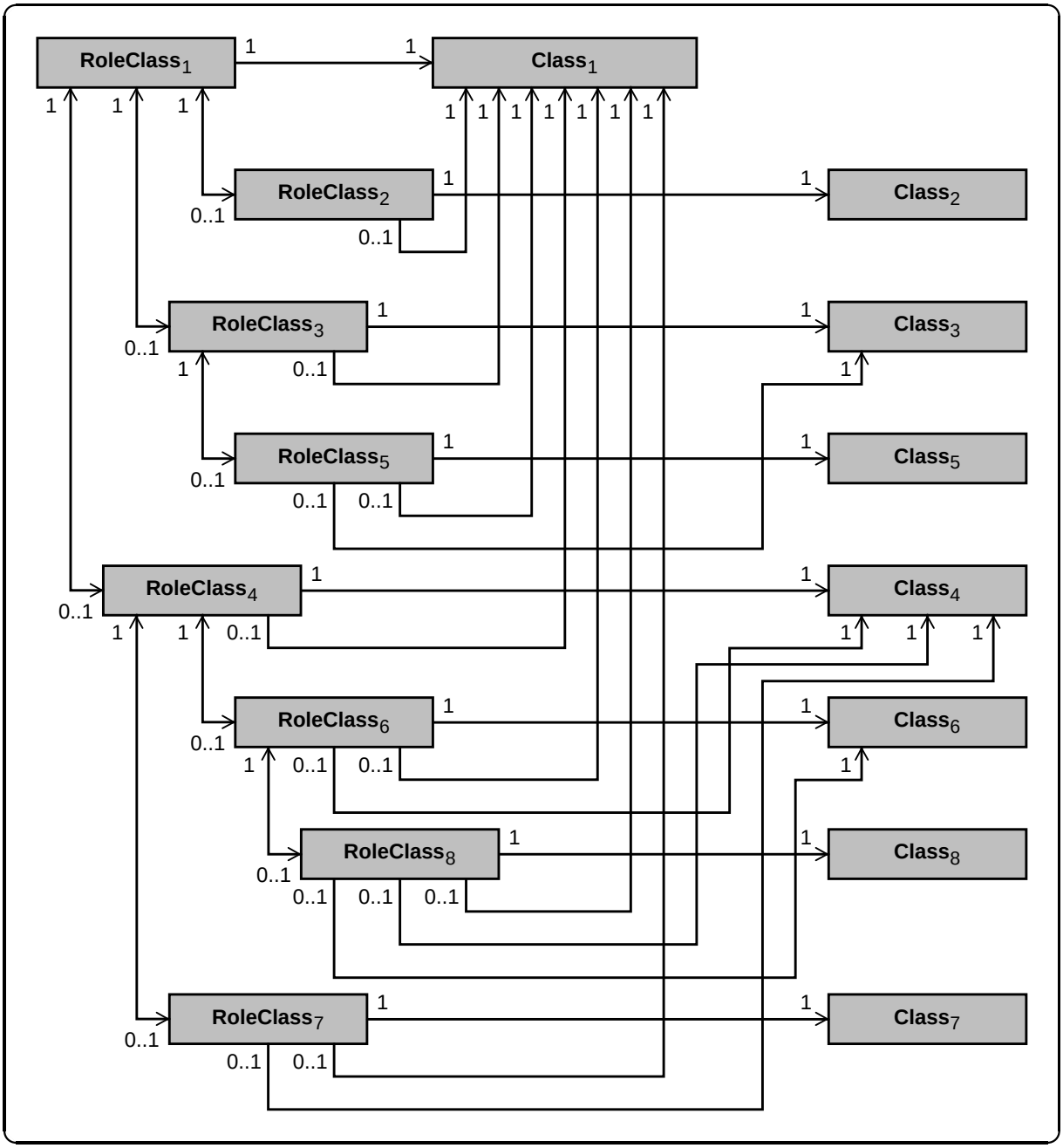


Abb. 44 Klassendiagramm des über Abb. 43 (S. 98) definierten Rollenmodells

Ein Objekt der Wurzerrolle besitzt eine feste Beziehung zu einem Objekt von **Class**$_1$. Mit der Erzeugung des Wurzelrollenobjekts wird auch das **Class**$_1$-Objekt erzeugt. Die Beziehung der beiden Objekte wird erst mit der Ausführung der `delete`-Operation auf dem **RoleClass**$_1$-Objekt wieder gelöscht. Zusätzlich kann ein Wurzelrollenobjekt jeweils eine Beziehung zu einem Objekt einer direkten Unterrolle besitzen. Diese Beziehung wird benötigt, um bei einem Löschen des Wurzelrollenobjekts alle vorhandenen Unterrollenobjekte zu löschen. Die Beziehung zu dem **RoleClass**$_i$-Objekt wird über die entsprechende `createRoleClass`$_i$-Operation aufgebaut, die auch die Erzeugung des Unterrollenobjekts vornimmt.

Ein Objekt einer Unterrollenklasse **RoleClass**$_k$ (k$>$1) verfügt neben der festen Beziehung zu dem Objekt der korrespondierenden Klasse über (feste) Beziehungen zu Objekten aller

Klassen **Class**$_j$, die direkte oder indirekte Vorgänger von **Class**$_k$ in der Baumstruktur des Rollenmodells sind. Alle diese Beziehungen werden innerhalb der `createRoleClass`$_k$-Operation der direkten Oberrollenklasse aufgebaut. Da diese Operation den einzigen Weg zur Erzeugung eines **RoleClass**$_k$-Objekts darstellt, ist garantiert, daß alle benötigten **Class**$_j$-Objekte bereits existieren. Die Beziehung zu dem direkten Oberrollenobjekt wird ebenfalls in der `createRoleClass`$_k$-Operation erzeugt. Die `delete`-Operation der **RoleClass**$_k$-Klasse ist dann wieder für den Abbau aller Beziehungen zuständig.

Die Attribute aus den Klassen **Class**$_1$ bis **Class**$_n$ werden in **RoleClass**$_1$ bis **RoleClass**$_n$ nicht übernommen. Stattdessen wird bei der Erzeugung eines **RoleClass**$_k$-Objekts ein Objekt der korrespondierenden Klasse erzeugt und eine Beziehung zwischen den beiden Objekten aufgebaut. Auch die Rümpfe der Operationen aus **Class**$_1$ bis **Class**$_n$ werden in den entsprechenden Operationen in **RoleClass**$_1$ bis **RoleClass**$_n$ nicht übernommen. Ein Aufruf einer Operation aus **RoleClass**$_k$ leitet die Kontrolle an die entsprechende Operation des zugehörigen **Class**$_k$-Objekts weiter.

Die Simulation des Vererbungskonzepts kann jetzt erreicht werden, indem eine **RoleClass**$_i$-Klasse an ihrer Schnittstelle nicht nur die Operationen aus ihrer korrespondierenden Klasse **Class**$_i$ anbietet, sondern auch alle Operationen aus den direkten[3] und indirekten Oberrollen. Für die in Abb. 43 (S. 98) dargestellte Rollenhierarchie besitzt beispielsweise **RoleClass**$_6$ alle Operationen, die in **Class**$_6$, **Class**$_4$ und **Class**$_1$ definiert sind.

Für die ***Sichtbarkeit***[4] der einzelnen Klassen des Rollenmodells und ihrer Bestandteile (Attribute, Konstruktoren, Operationen) sollen die folgenden Regeln gelten:

- Die Klassen **Class**$_1$ bis **Class**$_n$ sowie **RoleClass**$_1$ bis **RoleClass**$_n$ eines Rollenmodells liegen innerhalb desselben Pakets.
- Die Klassen **Class**$_1$ bis **Class**$_n$ sind nur *paketintern* sichtbar. Alle Attribute sind **`private`**, um die Einhaltung des Geheimniskonzepts zu gewährleisten.
- Die Klassen **RoleClass**$_1$ bis **RoleClass**$_n$ sollen außerhalb ihres Pakets sichtbar sein und sind daher als **`public`** definiert. Auf die Sichtbarkeit der Attribute, Konstruktoren und Methoden wird bei der Vorstellung der Struktur der Wurzel- und Unterrollen näher eingegangen.

Eine Applikation, die das Rollenmodell verwenden will, sollte einem anderen Paket zugeordnet werden. Damit ist sichergestellt, daß diese Applikation nur auf die Klassen **RoleClass**$_1$ bis **RoleClass**$_n$ zugreifen kann. Die interne Struktur des Rollenmodells bleibt ihr damit verborgen.

Die Aufgabe der Klassen **RoleClass**$_1$ bis **RoleClass**$_n$ besteht nun darin, den Anwendern des Rollenmodells die geforderte Schnittstelle zur Verfügung zu stellen. Zusätzlich ist die Konsistenz der Beziehungen der Rollenobjekte untereinander und zu den korrespondierenden Objekten zu gewährleisten. Bezüglich der internen Struktur ist zwischen der Wurzelrolle **RoleClass**$_1$ und den Unterrollen **RoleClass**$_2$ bis **RoleClass**$_n$ zu unterscheiden.

[3] Bei der zugrunde gelegten Einfachvererbung kann es natürlich nur eine direkte Oberrolle geben.

[4] Zur Spezifikation der Sichtbarkeit wird die JAVA-Semantik verwendet, da die Implementierung auch in JAVA ausgeführt wurde.

Die Wurzelrolle besitzt den folgenden Aufbau:

- Konstruktoren:
  Für jeden öffentlichen (d.h. **`public`**) Konstruktor aus $\mathbf{Class}_1$ wird ein **`public`**-Konstruktor in $\mathbf{RoleClass}_1$ bereitgestellt, der dieselben Parametertypen besitzt. Andere Konstruktoren aus $\mathbf{Class}_1$ besitzen in $\mathbf{RoleClass}_1$ keinen korrespondierenden Konstruktor. Liegt in $\mathbf{Class}_1$ kein öffentlicher Konstruktor vor, dann erhält $\mathbf{RoleClass}_1$ einen expliziten[5] parameterlosen Konstruktor. Ein $\mathbf{RoleClass}_1$-Konstruktor erzeugt das korrespondierende Objekt, indem er denjenigen Konstruktor aus $\mathbf{Class}_1$ aufruft, der dieselbe Signatur wie der $\mathbf{RoleClass}_1$-Konstruktor besitzt. Außerdem baut er die Verbindung zu dem korrespondierenden Objekt auf.
- Löschen des Wurzelrollenobjekts:
  Die `delete`-Operation ist **`public`** und hat die Aufgabe, zunächst (rekursiv) alle existierenden Unterrollenobjekte zu löschen, bevor sie das eigene Rollenobjekt als gelöscht markiert. Falls das Wurzelrollenobjekt bereits als gelöscht markiert ist, wird eine `RoleDeleted`-Exception erzeugt.
  Da JAVA keine Destruktoren kennt, wird ein Objekt erst dann zum Löschen für den Garbage Collector freigegeben, wenn die letzte Objektreferenz gelöscht wurde. Daher ist es notwendig, zwischen einem Rollenobjekt und seinem korrespondierenden Objekt zu unterscheiden. Ob und wann Referenzen für Rollenobjekte gelöscht werden, unterliegt der Kontrolle der Applikation, die ein Rollenobjekt erzeugt hat. Hier ist es prinzipiell möglich, die Referenz auf ein Rollenobjekt beliebig oft zu kopieren, wodurch es schwierig werden kann, ein Rollenobjekt durch das Löschen aller Referenzen freizugeben. Um zumindest ein logisches Löschen zu ermöglichen, wird über einen `delete`-Aufruf das Rollenobjekt intern als gelöscht markiert. Alle folgenden Operationsaufrufe führen dann zu einer `RoleDeleted`-Exception. Für das korrespondierende Objekt ist dagegen ein Löschen aller Referenzen möglich, da diese Referenzen ausschließlich innerhalb des Rollenobjekts sowie der eventuell existierenden Unterrollenobjekte verwaltet werden (vgl. Abb. 44 (S. 99)).
- Operationen zur Erzeugung von Unterrollen:
  $\mathbf{Class}_k$ sei ein *direkter* Nachfolger von $\mathbf{Class}_1$ aus der Baumstruktur zur Definition der Rollenhierarchie. Für jeden öffentlichen Konstruktor in $\mathbf{Class}_k$ existiert dann eine **`public`**-Operation $\texttt{createRoleClass}_k$, die die Parametertypen des Konstruktors übernimmt. Sollte $\mathbf{Class}_1$ keinen öffentlichen Konstruktor besitzen, wird eine parameterlose $\texttt{createRoleClass}_k$-Operation erzeugt. Eine $\texttt{createRoleClass}_k$-Operation überprüft als erstes, ob das Wurzelrollenobjekt als gelöscht markiert ist. Danach folgt der Test, ob die angeforderte Unterrolle bereits existiert. Auf den ersten Fehlerfall wird mit einer `RoleDeleted`-Exception, auf den zweiten mit einer `SubroleExists`-Exception reagiert. Im Nichtfehlerfall wird das Unterrollenobjekt durch den Aufruf des entsprechenden Konstruktors der Unterrollenklasse erzeugt, die Beziehung zu dem Unterrollenobjekt hergestellt und als Ergebnis die Referenz auf das neue Unterrollenobjekt an die aufrufende Applikation zurückgegeben.
- Operationen aus der korrespondierenden Klasse:
  Für jede öffentliche Operation aus $\mathbf{Class}_1$ wird eine Operation mit gleicher Signatur in $\mathbf{RoleClass}_1$ zur Verfügung gestellt. Zu den anderen Operationen aus $\mathbf{Class}_1$ gibt es in

[5] Explizit soll in diesem Zusammenhang bedeuten, daß tatsächlich ein Konstruktor erzeugt wird und nicht der von JAVA standardmäßig vorhandene, parameterlose Konstruktor benutzt wird.

RoleClass$_1$ keine entsprechenden Operationen. Eine Operation prüft zunächst wieder, ob das Wurzelrollenobjekt nicht bereits als gelöscht markiert ist. Im Erfolgsfall wird die entsprechende Operation auf dem korrespondierenden Objekt aufgerufen, im Fehlerfall eine `RoleDeleted`-Exception erzeugt.

- Operationen zum Testen, ob jeweils ein bestimmtes Unterrollenobjekt existiert:
  Für jeden direkten Nachfolger **Class**$_k$ von **Class**$_1$ aus der Rollenhierarchie wird eine `has-SubroleClass`$_k$-Operation definiert, deren Aufruf als Ergebnis einen booleschen Wert zurückliefert.
- Test, ob das Rollenobjekt noch existiert:
  Die Operation `isDeleted` liefert als Ergebnis **`true`** zurück, wenn das Wurzelrollenobjekt als gelöscht markiert wurde.

Eine Unterrolle **RoleClass**$_k$ (k=2,. . .,n) besitzt die folgende Struktur:

- Konstruktoren:
  Für jeden öffentlichen Konstruktor aus **Class**$_k$ wird in **RoleClass**$_k$ ein nur paketintern sichtbarer Konstruktor erzeugt. Andere Konstruktoren aus **Class**$_k$ werden in **RoleClass**$_k$ nicht berücksichtigt. Der Konstruktor übernimmt die Parametertypen des entsprechenden **Class**$_k$-Konstruktors. Die Liste der formalen Parameter wird um eine Referenz auf die direkte Oberrolle sowie um Referenzen auf die korrespondierenden Objekte aller direkten und indirekten Oberrollen ergänzt. Falls in **Class**$_k$ kein öffentlicher Konstruktor vorliegt, wird für **RoleClass**$_k$ ein Konstruktor generiert, der nur die ergänzenden Referenzen als Parameter besitzt. Ein **RoleClass**$_k$-Konstruktor erzeugt das korrespondierende Objekt, indem er den entsprechenden Konstruktor aus **Class**$_k$ aufruft. Zusätzlich baut er die Beziehungen zu dem korrespondierenden Objekt, dem Oberrollenobjekt und allen korrespondierenden Objekten der direkten und indirekten Oberrollen auf.
- Operationen zur Erzeugung von Unterrollen:
  Class$_j$ sei ein *direkter* Nachfolger von **Class**$_k$ aus der Baumstruktur zur Definition der Rollenhierarchie. Für jeden öffentlichen Konstruktor in **Class**$_j$ existiert dann eine **`public`**-Operation `createRoleClass`$_j$, die die Parametertypen dieses Konstruktors übernimmt. Die restlichen Konstruktoren aus **Class**$_j$ besitzen in **RoleClass**$_j$ kein Gegenstück. Wenn **Class**$_j$ keinen öffentlichen Konstruktor besitzt, wird für **RoleClass**$_k$ die Erzeugung einer parameterlosen `createRoleClass`$_j$-Operation vorgenommen. Die interne Struktur einer `createRoleClass`$_j$-Operation ist identisch zu der Struktur einer `createRoleClass`-Operation aus der Wurzelrolle **RoleClass**$_1$ (vgl. S. 101).
- Operationen aus der korrespondierenden Klasse:
  Für jede öffentliche Operation aus **Class**$_k$ wird eine **`public`**-Operation mit gleicher Signatur zur Verfügung gestellt. Andere Operationen aus **Class**$_k$ finden dagegen keine Berücksichtigung. Eine Operation besitzt dieselbe Struktur wie eine entsprechende Operation der Wurzelrolle (vgl. S. 101).
- Operationen aus übergeordneten Klassen:
  Class$_i$ sei ein direkter oder indirekter Vorgänger von **Class**$_k$ aus der Baumstruktur zur Definition der Rollenhierarchie, d.h. ein Element des Pfades von **Class**$_k$[6] nach **Class**$_1$. Für jede öffentlich sichtbare Operation aus **Class**$_i$ enthält **RoleClass**$_k$ eine **`public`**-Operation

[6] Da es um die übergeordneten Klassen geht, wird das Element **Class**$_k$ aus dem Pfad nicht berücksichtigt.

mit derselben Signatur. Die weiteren Operationen aus **Class**$_i$ besitzen keine entsprechenden Operationen in **RoleClass**$_k$. Existieren auf dem Pfad von **Class**$_k$ nach **Class**$_1$ mehrere Operationen mit derselben Signatur, wird nur die als erstes auf diesem Pfad gefundene Operation in **RoleClass**$_k$ berücksichtigt. Auf diesem Weg läßt sich eine Operation einer Oberrolle durch eine entsprechende Operation einer Unterrolle verdecken. Eine Operation besitzt dieselbe Struktur wie eine entsprechende Operation der Wurzelrolle (vgl. S. 101).

- Löschen des Unterrollenobjekts:
  Analog zur Wurzelrolle (vgl. S. 101) besitzt eine Unterrolle **RoleClass**$_k$ eine **`public`**-Operation `delete`, die zunächst (rekursiv) alle vorhandenen Unterrollenobjekte des **RoleClass**$_k$-Objekts löscht und dann das **RoleClass**$_k$-Objekt selbst als gelöscht markiert.
- Operationen zum Testen, ob ein bestimmtes Unterrollenobjekt bereits existiert:
  Für jeden direkten Nachfolger **Class**$_i$ von **Class**$_k$ aus der Rollenhierarchie wird eine `hasSubroleClass`$_i$-Operation definiert, deren Aufruf als Ergebnis einen booleschen Wert zurückliefert.
- Test, ob das Unterrollenobjekt noch existiert:
  Die Operation `isDeleted` liefert als Ergebnis **`true`** zurück, wenn das Unterrollenobjekt als gelöscht markiert wurde.

Die in den **`throws`**-Klauseln der Konstruktoren und Operationen der korrespondierenden Klassen aufgeführten Exception-Klassen werden bei den entsprechenden Konstruktoren, `createRole`-Operationen und den sonstigen Operationen übernommen. Sie werden um die Exception-Klassen ergänzt, die für die Konsistenzerhaltung des Rollenmodells definiert wurden: `RoleDeleted` und `SubroleExists` bei den `createRole`-Operationen sowie `RoleDeleted` bei den sonstigen Operationen. Eine Ausnahme bilden lediglich die `isDeleted`-Operation und die Klassenoperationen. Hier ist jeweils eine `RoleDeleted`-Exception semantisch nicht sinnvoll.

5.2.3 Beispiel: Universitätsverwaltung

Für die in Abb. 9 (S. 16) vorgestellte Rollenhierarchie einer Universitätsverwaltung sind in Abb. 45 (S. 104) konkrete Beispiele für die korrespondierenden Klassen angegeben. Zwischen diesen Klassen bestehen keine Assoziationen. Es ist aber natürlich erlaubt, daß sie Assoziationen zu Klassen besitzen, die nicht direkt zum Rollenmodell gehören. In Abb. 45 (S. 104) ist **Adresse** ein Beispiel für eine derartige Klasse. Wenn die Klasse **Adresse** in demselben Paket wie die korrespondierenden Klassen liegt, dann muß sie als **`public`** definiert werden, damit sie außerhalb des Pakets nutzbar ist. Dies ist erforderlich, da davon ausgegangen wurde, daß eine das Rollenmodell nutzende Applikation in einem anderen Paket liegt als das Rollenmodell selbst.

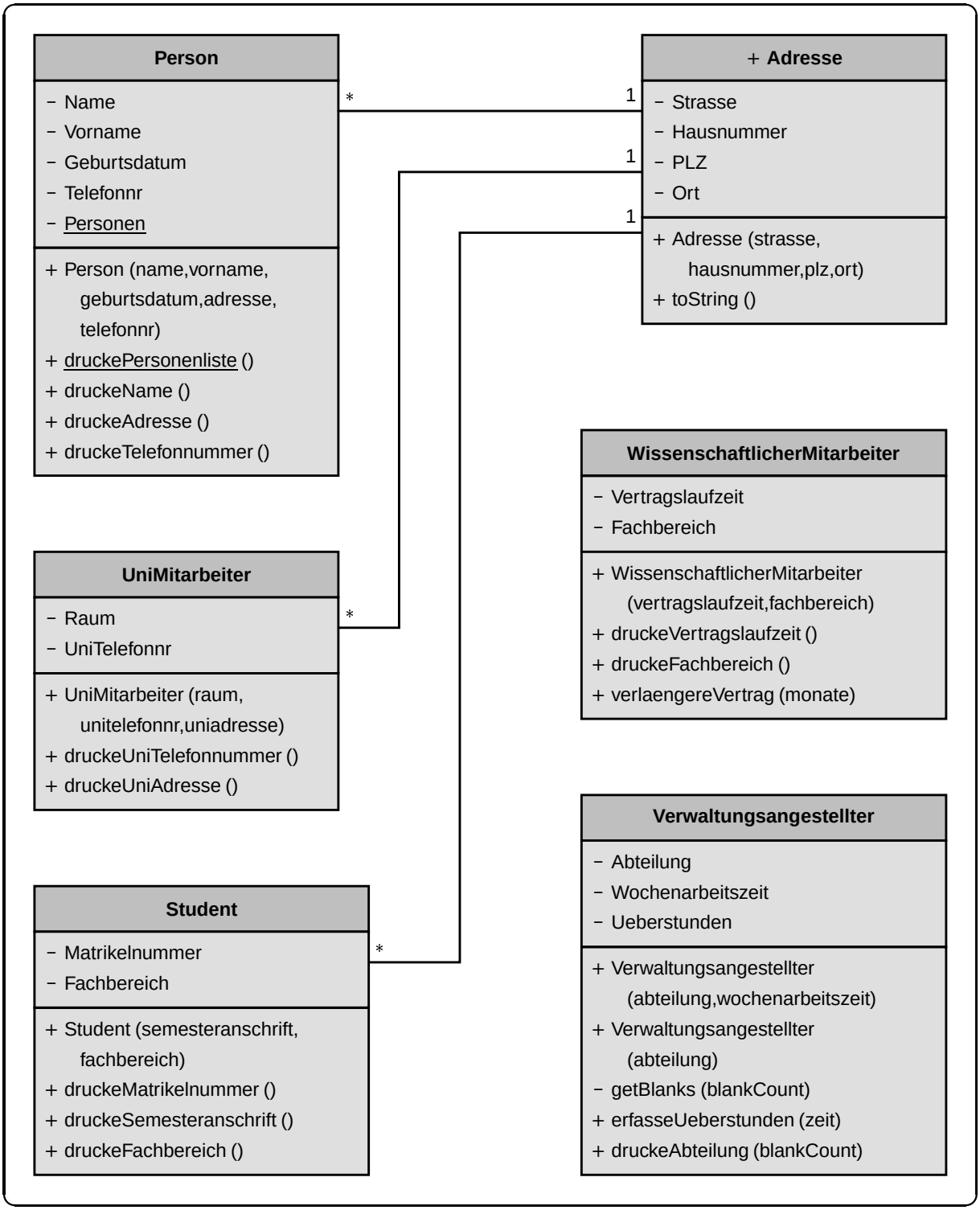


Abb. 45 Klassendiagramm für die korrespondierenden Klassen der Rollenhierarchie aus Abb. 9 (S. 16)

In den Abbildungen Abb. 46 bis Abb. 50 (S. 109) ist die Struktur der Rollenklassen zu sehen. Neben den Operationen sind die Attribute angegeben, die zur Realisierung der in Abb. 44 (S. 99) dargestellten Struktur eines Rollenmodells benötigt werden. Als Attributnamen wurden diejenigen verwendet, die der in Abschnitt 5.10 (S. 197) beschriebene Rollenmodellgenerator erzeugt. Um den Aufbau der Rollenklassen leichter nachvollziehen zu können, sind in den Abbildungen auch jeweils die zugehörigen korrespondierenden Klassen angegeben, wobei die Attribute weggelassen wurden.

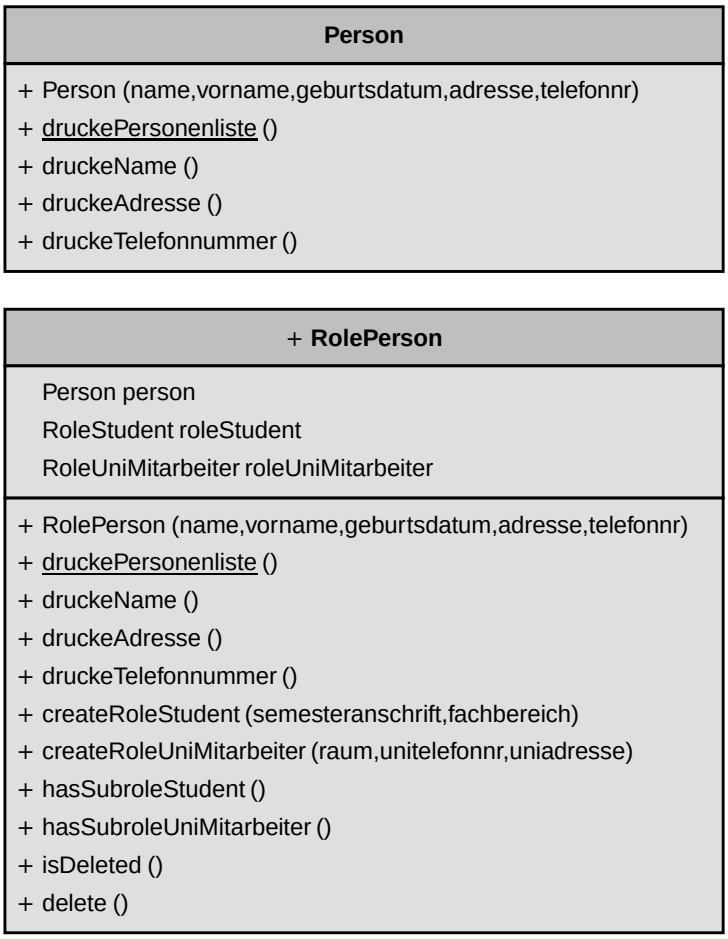


Abb. 46 Struktur der Klasse **RolePerson**

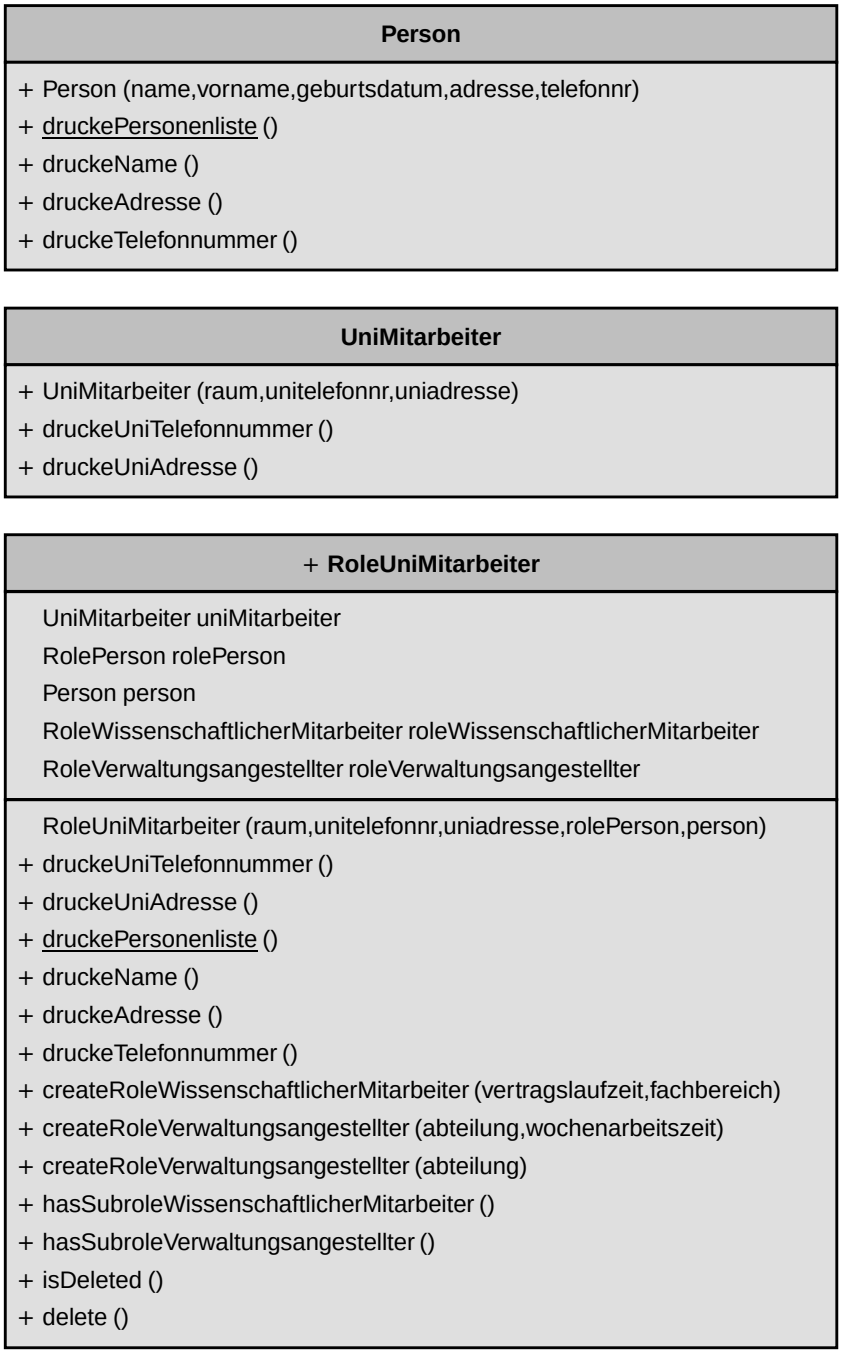


Abb. 47 Struktur der Klasse **RoleUniMitarbeiter**

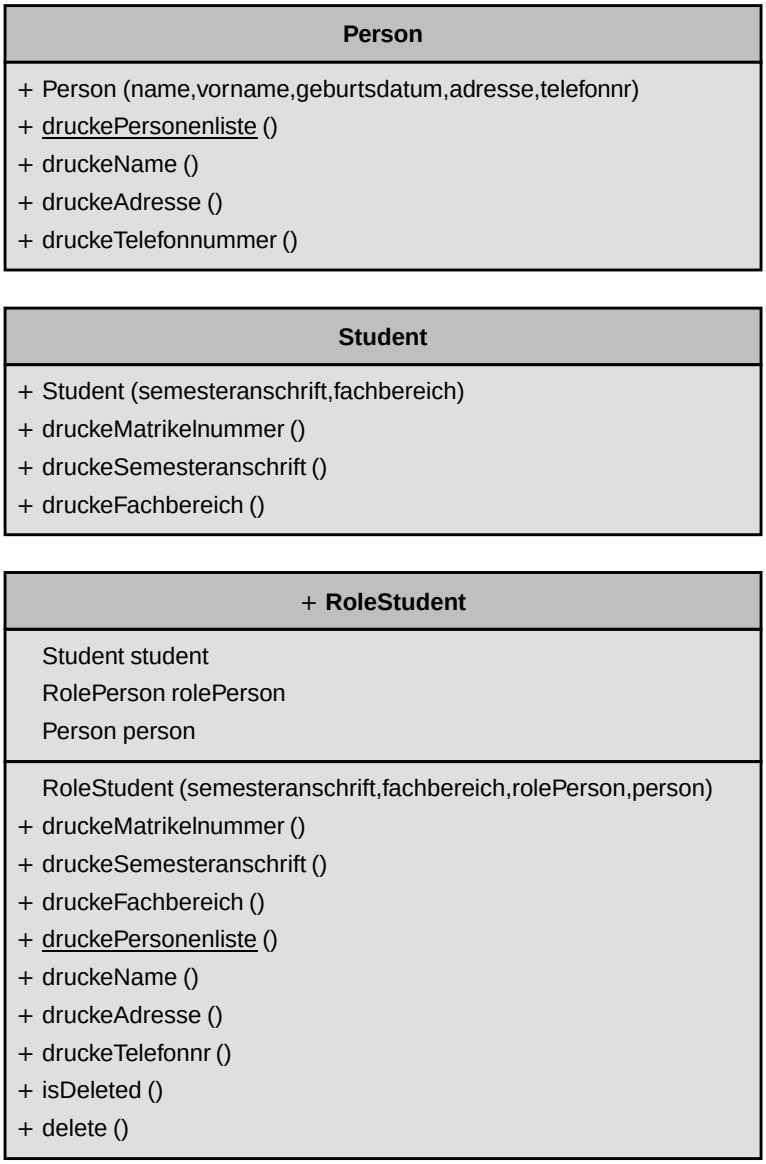


Abb. 48 Struktur der Klasse **RoleStudent**

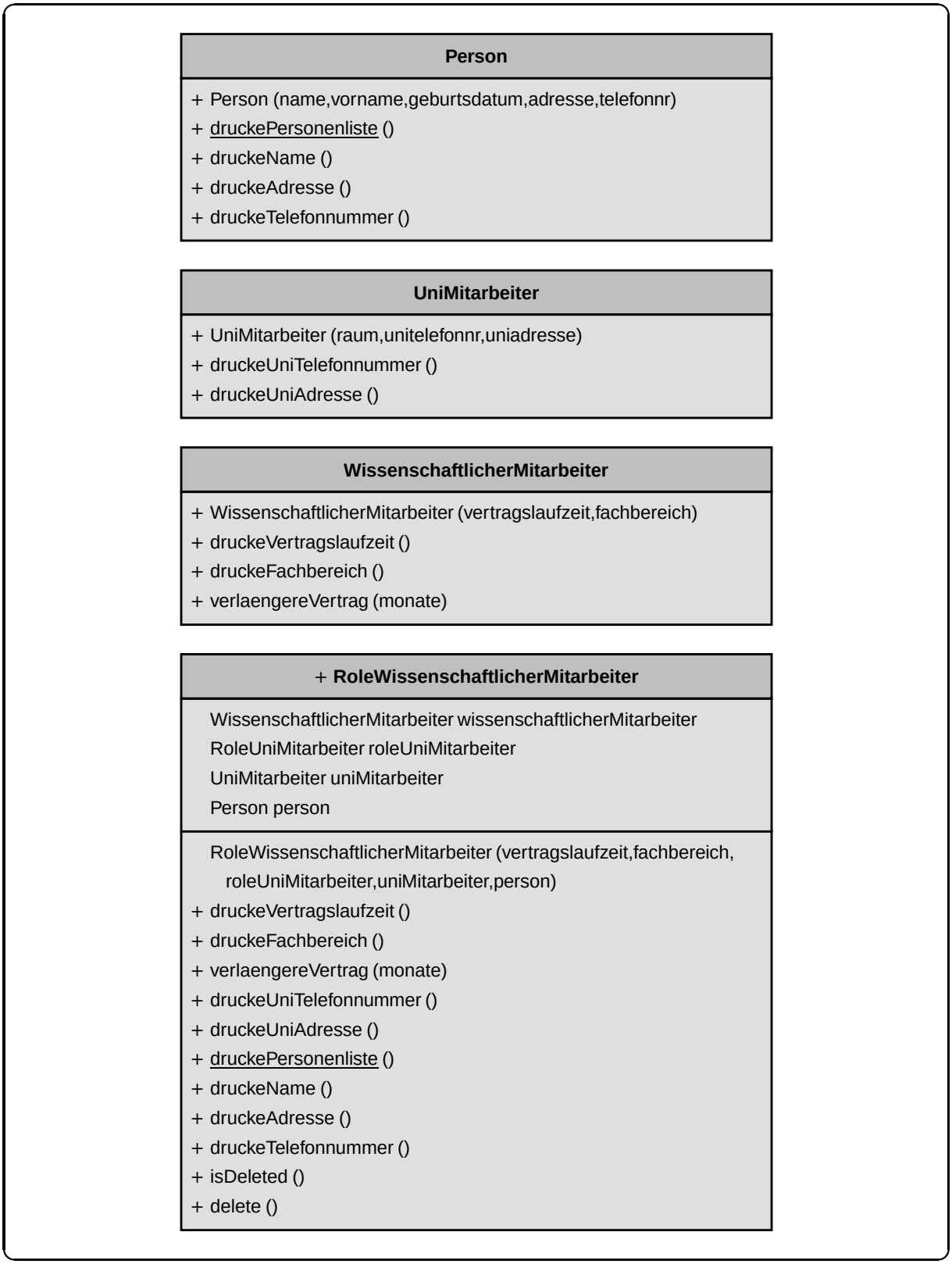


Abb. 49 Struktur der Klasse **RoleWissenschaftlicherMitarbeiter**

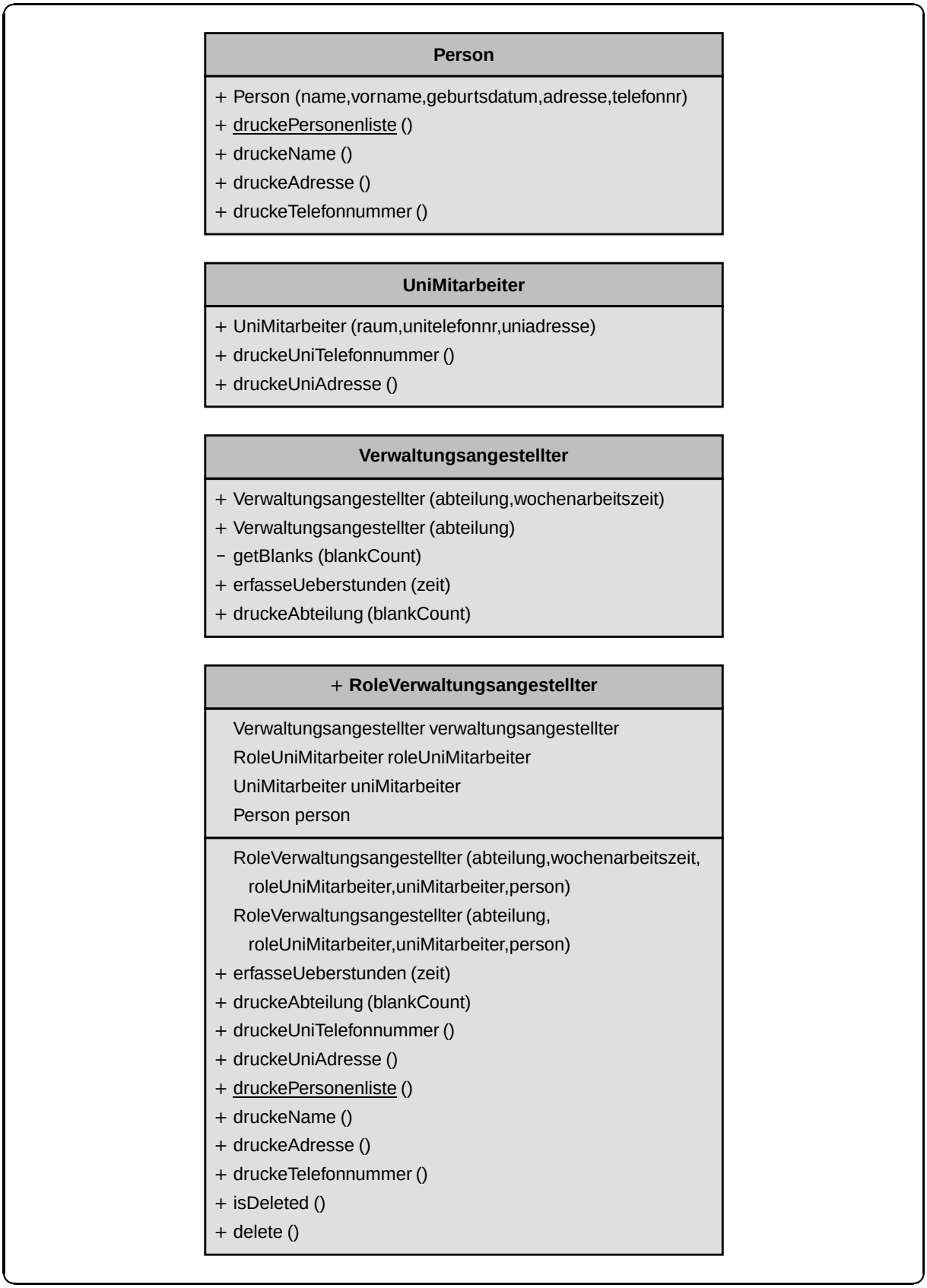


Abb. 50 Struktur der Klasse **RoleVerwaltungsangestellter**

5.2.4 Generierung des Rollenmodells

Die Generierung eines Rollenmodells besteht aus drei Schritten:

- Die Definition der Baumstruktur für die Rollenhierarchie.
- Die Implementierung der zu den Rollen korrespondierenden Klassen.
- Die Generierung selbst.

Für die Definition der Rollenhierarchie ist eine Datei zu erzeugen, deren Inhalt den folgenden Syntaxregeln genügen muß:

```
RoleModelDescriptionSI ::= rolemodel$ RootRoleRelationship+
                           RoleRelationship* $rolemodel
RootRoleRelationship   ::= root$ RoleIdentifier
                           hasroles$ RoleIdentifier+ $root
RoleRelationship       ::= role$ RoleIdentifier
                           hasroles$ RoleIdentifier+ $role
RoleIdentifier         ::= UpperCaseLetter
                           ⟨ UpperCaseLetter | LowerCaseLetter ⟩*
UpperCaseLetter        ::= { A,B,...,Z }
LowerCaseLetter        ::= { a,b,...,z }
```

Das Akronym *SI* in *RoleModelDescriptionSI* steht für *SingleInheritance*. Über die obige Grammatik lassen sich beliebige Graphen definieren. Der Parser übernimmt dann die Aufgabe zu prüfen, ob auch wirklich eine Baumstruktur vorliegt und die in *RootRoleRelationship* definierte Wurzelrolle tatsächlich die Wurzel des Baumes darstellt.

Für jeden in *RoleModelDescriptionSI* auftretenden *RoleIdentifier* ist eine Klasse **RoleIdentifier** in einer Datei `RoleIdentifier.java` zu implementieren. Diese Klassen stellen die zu den Rollenklassen **RoleClass**$_i$ korrespondierenden Klassen **Class**$_i$ dar.

Prg. 3 zeigt den Inhalt der Beschreibungsdatei für die Universitätsverwaltung aus Abb. 9 (S. 16). Der Quelltext der Klassen **Person** und **UniMitarbeiter** ist in Prg. 4 (S. 111) und Prg. 5 (S. 112) enthalten. Die generierte Klasse **RolePerson** ist in Prg. 6 (S. 113) bis Prg. 8 (S. 115) zu sehen, die generierte Klasse **RoleUniMitarbeiter** in Prg. 9 (S. 116) bis Prg. 12 (S. 119).[7]

```
1:  rolemodel$
2:  root$ Person hasroles$ Student UniMitarbeiter $root
3:  role$ UniMitarbeiter
4:        hasroles$ WissenschaftlicherMitarbeiter $role
5:  role$ UniMitarbeiter
6:        hasroles$ Verwaltungsangestellter $role
7:  $rolemodel
```

Prg. 3 Beschreibungsdatei der Universitätsverwaltung

[7] Die dargestellten Klassen entsprechen bis auf partielle Änderungen des Zeilenumbruchs den generierten Klassen.

```
 1: package roleModel.singleInheritance;
 2:
 3: import java.util.Vector;
 4:
 5: class Person {
 6:   private String name;
 7:   private String vorname;
 8:   private String geburtsdatum;
 9:   private Adresse adresse;
10:   private String telefonnr;
11:   private static Vector personen = new Vector();
12:
13:   public Person ( String name, String vorname, String geburtsdatum,
14:                   Adresse adresse, String telefonnr ) {
15:     this.name = name;
16:     this.vorname = vorname;
17:     this.geburtsdatum = geburtsdatum;
18:     this.adresse = adresse;
19:     this.telefonnr = telefonnr;
20:     personen.addElement ( this );
21:   }
22:
23:   public static void druckePersonenliste () {
24:     for ( int i=0; i<personen.size(); i++ ) {
25:         Person p = (Person) personen.elementAt(i);
26:         p.druckeName();
27:     } // for
28:   } // druckePersonenliste
29:
30:   public void druckeName () {
31:     System.out.println ( "   " + name + ", " + vorname );
32:   } // druckeName
33:
34:   public void druckeAdresse () {
35:     System.out.println ( "Privatadresse:" );
36:     System.out.println ( adresse );
37:   } // druckeAdresse
38:
39:   public void druckeTelefonnummer () {
40:     System.out.println ( "Privatnummer = "  + telefonnr );
41:   } // druckeTelefonnummer
42:
43: } // Person
```

Prg. 4 Die Klasse **Person**

```
 1: package roleModel.singleInheritance;
 2:
 3: class UniMitarbeiter {
 4:   private String raum;
 5:   private String uniTelefonnr;
 6:   private Adresse uniAdresse;
 7:
 8:   public UniMitarbeiter ( String raum, String uniTelefonnr,
 9:                           Adresse uniAdresse ) {
10:     this.raum = raum;
11:     this.uniTelefonnr = uniTelefonnr;
12:     this.uniAdresse = uniAdresse;
13:   } // UniMitarbeiter
14:
15:   public void druckeUniTelefonnummer () {
16:     System.out.println ( "Telefon Universitaet: " + uniTelefonnr );
17:   } // druckeUniTelefonnummer
18:
19:   public void druckeUniAdresse () {
20:     System.out.println ( "Universitaetsadresse\n" + raum + "\n" +
21:                          uniAdresse );
22:   } //
23:
24: } // UniMitarbeiter
```

Prg. 5 Die Klasse **UniMitarbeiter**

```
 1: package roleModel.singleInheritance;
 2:
 3: import scanner.*;
 4:
 5: public class RolePerson {
 6:
 7:   Person person = null;
 8:
 9:   RoleStudent roleStudent = null;
10:   RoleUniMitarbeiter roleUniMitarbeiter = null;
11:
12:   public RolePerson ( String name, String vorname,
13:            String geburtsdatum, Adresse adresse,
14:            String telefonnr ) {
15:     person = new Person ( name,vorname,geburtsdatum,adresse,
16:        telefonnr );
17:   }
18:
19:   public static void druckePersonenliste () {
20:     Person.druckePersonenliste ();
21:   }
22:
23:   public void druckeName ()
24:          throws RoleDeleted {
25:     if ( person==null )
26:        throw new RoleDeleted ( "Person" );
27:     person.druckeName ();
28:   }
29:
30:   public void druckeAdresse ()
31:          throws RoleDeleted {
32:     if ( person==null )
33:        throw new RoleDeleted ( "Person" );
34:     person.druckeAdresse ();
35:   }
36:
37:   public void druckeTelefonnummer ()
38:          throws RoleDeleted {
39:     if ( person==null )
40:        throw new RoleDeleted ( "Person" );
41:     person.druckeTelefonnummer ();
42:   }
```

Prg. 6 Die Rolle **RolePerson** – Teil 1

```
43:
44:    public RoleStudent createRoleStudent
45:             ( Adresse semesteranschrift, String fachbereich )
46:           throws SubroleExists,RoleDeleted {
47:      if ( person==null )
48:         throw new RoleDeleted ( "Person" );
49:      if ( roleStudent!=null )
50:         throw new SubroleExists ( "Student" );
51:      roleStudent = new RoleStudent ( semesteranschrift,
52:         fachbereich,this,person );
53:      return roleStudent;
54:    }
55:
56:    public RoleUniMitarbeiter createRoleUniMitarbeiter
57:             ( String raum, String uniTelefonnr,
58:               Adresse uniAdresse )
59:           throws SubroleExists,RoleDeleted {
60:      if ( person==null )
61:         throw new RoleDeleted ( "Person" );
62:      if ( roleUniMitarbeiter!=null )
63:         throw new SubroleExists ( "UniMitarbeiter" );
64:      roleUniMitarbeiter = new RoleUniMitarbeiter ( raum,
65:         uniTelefonnr,uniAdresse,this,person );
66:      return roleUniMitarbeiter;
67:    }
68:
69:    public boolean hasSubroleStudent()
70:           throws RoleDeleted {
71:      if ( person==null )
72:         throw new RoleDeleted ( "Person" );
73:      return roleStudent!=null &&
74:             roleStudent.student!=null;
75:    }
76:
77:    public boolean hasSubroleUniMitarbeiter()
78:           throws RoleDeleted {
79:      if ( person==null )
80:         throw new RoleDeleted ( "Person" );
81:      return roleUniMitarbeiter!=null &&
82:             roleUniMitarbeiter.uniMitarbeiter!=null;
83:    }
```

Prg. 7 Die Rolle **RolePerson** – Teil 2

```
84:
85:    public boolean isDeleted () {
86:      return person == null;
87:    }
88:
89:    public void delete () throws RoleDeleted {
90:      if ( person==null )
91:         throw new RoleDeleted ( "Person" );
92:      person = null;
93:      if ( roleStudent != null )
94:         roleStudent.delete();
95:      if ( roleUniMitarbeiter != null )
96:         roleUniMitarbeiter.delete();
97:    }
98:
99:  }
```

Prg. 8 Die Rolle **RolePerson** – Teil 3

```
 1: package roleModel.singleInheritance;
 2:
 3: import scanner.*;
 4:
 5: public class RoleUniMitarbeiter {
 6:
 7:   UniMitarbeiter uniMitarbeiter = null;
 8:
 9:   RolePerson rolePerson = null;
10:
11:   Person person = null;
12:
13:   RoleWissenschaftlicherMitarbeiter
14:     roleWissenschaftlicherMitarbeiter = null;
15:   RoleVerwaltungsangestellter roleVerwaltungsangestellter = null;
16:
17:   RoleUniMitarbeiter ( String raum, String uniTelefonnr,
18:       Adresse uniAdresse, RolePerson rolePerson, Person person ) {
19:     uniMitarbeiter = new UniMitarbeiter ( raum,uniTelefonnr,
20:        uniAdresse );
21:     this.rolePerson = rolePerson;
22:     this.person = person;
23:   }
24:
25:   public void druckeUniTelefonnummer ()
26:          throws RoleDeleted {
27:     if ( uniMitarbeiter==null )
28:        throw new RoleDeleted ( "UniMitarbeiter" );
29:     uniMitarbeiter.druckeUniTelefonnummer ();
30:   }
31:
32:   public void druckeUniAdresse ()
33:          throws RoleDeleted {
34:     if ( uniMitarbeiter==null )
35:        throw new RoleDeleted ( "UniMitarbeiter" );
36:     uniMitarbeiter.druckeUniAdresse ();
37:   }
38:
39:   public static void druckePersonenliste () {
40:     Person.druckePersonenliste ();
41:   }
42:
43:   public void druckeName ()
44:          throws RoleDeleted {
45:     if ( person==null )
46:        throw new RoleDeleted ( "Person" );
47:     person.druckeName ();
48:   }
```

Prg. 9 | Die Rolle **RoleUniMitarbeiter** – Teil 1

```
49:
50:    public void druckeAdresse ()
51:           throws RoleDeleted {
52:      if ( person==null )
53:         throw new RoleDeleted ( "Person" );
54:      person.druckeAdresse ();
55:    }
56:
57:    public void druckeTelefonnummer ()
58:           throws RoleDeleted {
59:      if ( person==null )
60:         throw new RoleDeleted ( "Person" );
61:      person.druckeTelefonnummer ();
62:    }
63:
64:    public RoleWissenschaftlicherMitarbeiter
65:           createRoleWissenschaftlicherMitarbeiter
66:             ( int vertragslaufzeit, String fachbereich )
67:           throws SubroleExists,RoleDeleted {
68:      if ( uniMitarbeiter==null )
69:         throw new RoleDeleted ( "UniMitarbeiter" );
70:      if ( roleWissenschaftlicherMitarbeiter!=null )
71:         throw new SubroleExists ( "WissenschaftlicherMitarbeiter" );
72:      roleWissenschaftlicherMitarbeiter
73:         = new RoleWissenschaftlicherMitarbeiter
74:         ( vertragslaufzeit,fachbereich,this,uniMitarbeiter,
75:           person );
76:      return roleWissenschaftlicherMitarbeiter;
77:    }
78:
79:    public RoleVerwaltungsangestellter
80:           createRoleVerwaltungsangestellter
81:             ( String abteilung, double wochenarbeitszeit )
82:           throws SubroleExists,RoleDeleted {
83:      if ( uniMitarbeiter==null )
84:         throw new RoleDeleted ( "UniMitarbeiter" );
85:      if ( roleVerwaltungsangestellter!=null )
86:         throw new SubroleExists ( "Verwaltungsangestellter" );
87:      roleVerwaltungsangestellter = new RoleVerwaltungsangestellter
88:         ( abteilung,wochenarbeitszeit,this,uniMitarbeiter,
89:           person );
90:      return roleVerwaltungsangestellter;
91:    }
```

Prg. 10 | Die Rolle **RoleUniMitarbeiter** – Teil 2

```
 92:
 93:    public RoleVerwaltungsangestellter
 94:           createRoleVerwaltungsangestellter
 95:             ( String abteilung )
 96:           throws SubroleExists,RoleDeleted {
 97:      if ( uniMitarbeiter==null )
 98:         throw new RoleDeleted ( "UniMitarbeiter" );
 99:      if ( roleVerwaltungsangestellter!=null )
100:         throw new SubroleExists ( "Verwaltungsangestellter" );
101:      roleVerwaltungsangestellter = new RoleVerwaltungsangestellter
102:         ( abteilung,this,uniMitarbeiter,person );
103:      return roleVerwaltungsangestellter;
104:    }
105:
106:    public boolean hasSubroleWissenschaftlicherMitarbeiter()
107:           throws RoleDeleted {
108:      if ( uniMitarbeiter==null )
109:         throw new RoleDeleted ( "UniMitarbeiter" );
110:      return roleWissenschaftlicherMitarbeiter!=null &&
111:      roleWissenschaftlicherMitarbeiter.wissenschaftlicherMitarbeiter
112:         !=null;
113:    }
114:
115:    public boolean hasSubroleVerwaltungsangestellter()
116:           throws RoleDeleted {
117:      if ( uniMitarbeiter==null )
118:         throw new RoleDeleted ( "UniMitarbeiter" );
119:      return roleVerwaltungsangestellter!=null &&
120:             roleVerwaltungsangestellter.verwaltungsangestellter
121:                !=null;
122:    }
123:
124:    public boolean isDeleted () {
125:      return uniMitarbeiter == null;
126:    }
```

Prg. 11 Die Rolle **RoleUniMitarbeiter** – Teil 3

```
127:
128:    public void delete () throws RoleDeleted {
129:      if ( uniMitarbeiter==null )
130:         throw new RoleDeleted ( "UniMitarbeiter" );
131:      uniMitarbeiter = null;
132:      person = null;
133:      if ( roleWissenschaftlicherMitarbeiter != null )
134:         roleWissenschaftlicherMitarbeiter.delete();
135:      if ( roleVerwaltungsangestellter != null )
136:         roleVerwaltungsangestellter.delete();
137:      rolePerson.roleUniMitarbeiter = null;
138:      rolePerson = null;
139:    }
140:
141:  }
```

Prg. 12 Die Rolle **RoleUniMitarbeiter** – Teil 4

Auf eine Darstellung der Rollen **RoleStudent**, **RoleWissenschaftlicherMitarbeiter** und **RoleVerwaltungsangestellter** wurde verzichtet, da sie strukturell mit der Rolle **RoleUniMitarbeiter** vergleichbar sind.

5.2.5 Assoziationen zwischen den Klassen $\mathbf{Class}_1$ bis $\mathbf{Class}_n$

Für das in Abschnitt 5.2.1 (S. 97) vorgestellte Rollenmodell wurde angenommen, daß zwischen den korrespondierenden Klassen keine Assoziationen bestehen. Dadurch ist es nicht möglich, in einer Operation von $\mathbf{Class}_i$ auf eine Operation von $\mathbf{Class}_j$ ($i \neq j$) zuzugreifen. Sollen beispielsweise für einen wissenschaftlichen Mitarbeiter seine Daten ausgegeben werden, so gehören dazu sinnvollerweise auch die Daten aus seiner UniMitarbeiter- und seiner Personenrolle. Um diese Eigenschaft zu realisieren, muß es möglich sein, zwischen korrespondierenden Klassen Assoziationen der Kardinalität 1 aufzubauen. Für eine Integration in das Rollenmodell sind bei der Definition einer Assoziation zwischen zwei Klassen $\mathbf{Class}_i$ und $\mathbf{Class}_j$ ($i \neq j$) drei Restriktionen einzuhalten:

- R1: $\mathbf{Class}_i$ ist ein direkter oder indirekter Vorgängerknoten von $\mathbf{Class}_j$ in der die Rollenhierarchie definierenden Baumstruktur.
- R2: Die Assoziation existiert nur in der Richtung von $\mathbf{Class}_j$ nach $\mathbf{Class}_i$.
- R3: Der Aufbau einer Beziehung zwischen zwei Objekten der Klassen $\mathbf{Class}_j$ und $\mathbf{Class}_i$ findet innerhalb eines jeden Konstruktors der Klasse $\mathbf{Class}_j$ statt, wobei die Referenz auf das $\mathbf{Class}_i$-Objekt als Parameter dem $\mathbf{Class}_j$-Konstruktor zu übergeben ist.

Die letzte Restriktion stellt sicher, daß die Beziehung zwischen den Objekten auf jeden Fall existiert, bevor eine Operation aufgerufen wird, die diese Beziehung benötigt. Die ersten beiden Restriktionen sind notwendig, um bei der Erzeugung der Rollenobjekte die Beziehungen zwischen den Objekten der korrespondierenden Klassen aufbauen zu können.

Abb. 51 zeigt ein Beispiel, in dem zwischen den korrespondierenden Klassen **Mitarbeiter** und **Angestellter** eine Assoziation besteht. Sie wird von der `ausgebenDaten`-Operation in **Angestellter** benutzt, um auf die entsprechende Operation in **Mitarbeiter** zuzugreifen.

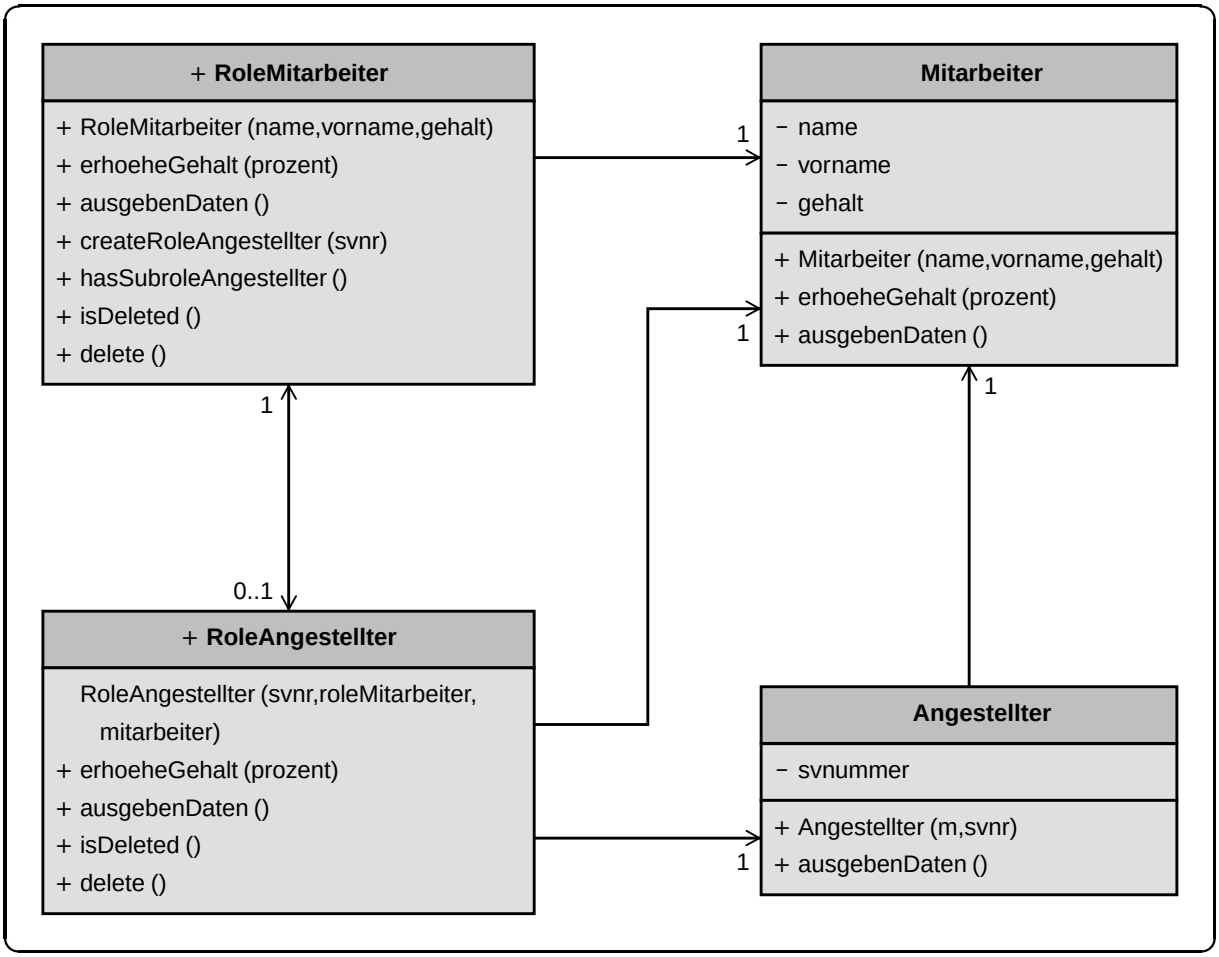


Abb. 51 | Assoziationen zwischen den korrespondierenden Klassen

Im Vergleich zu dem Rollenmodell aus Abschnitt 5.2.1 (S. 97) sind folgende Änderungen vorzunehmen:

- `createRole`-Operationen:
  Falls der Konstruktor der korrespondierenden Klasse **Class**$_j$ als Parameter eine Referenz auf die Klasse **Class**$_i$ besitzt, dann darf die `createRoleClass`$_j$-Operation in **RoleClass**$_i$ den zugehörigen Parameter nicht übernehmen, da **Class**$_i$ als korrespondierende Klasse nur paketintern sichtbar ist. In Abb. 51 hat der Konstruktor der Klasse **Angestellter** (=**Class**$_j$) als Parameter eine Referenz `m`, die die Beziehung zu dem **Mitarbeiter**-Objekt aufbaut. Die `createRoleAngestellter`-Operation der Klasse **RoleMitarbeiter** (=**RoleClass**$_i$) besitzt diesen Parameter nicht mehr.
- Konstruktoren der Unterrollen **RoleClass**$_j$ (d.h. $j>1$):
  Bei der Erzeugung eines Konstruktors müssen aus der Parameterliste des entsprechenden Konstruktors aus der korrespodierenden Klasse zunächst alle formalen Parameter entfernt werden, die Referenzen auf korrespondierende Klassen **Class**$_i$ ($j \neq i$) darstellen. Der Kon-

struktor der Klasse **Angestellter** (=**RoleClass**$_j$) aus Abb. 51 (S. 120) enthält den Parameter `m`, der die Beziehung zu dem **Mitarbeiter**-Objekt (=**Class**$_i$) herstellt. Dieser Parametername tritt in dem Konstruktor `RoleAngestellter` nicht mehr auf.

Da ein Konstruktor aber standardmäßig in seiner formalen Parameterliste Referenzen auf alle korrespondierenden Klassen der direkten und indirekten Oberrollen enthält, stellen die Restriktionen R1 und R2 sicher, daß der Konstruktor auch einen **Class**$_i$-Parameter besitzt. Außerdem muß aufgrund der Konsistenzregeln für das Rollenmodell bereits ein **Class**$_i$-Objekt existieren: **RoleClass**$_i$ ist eine (direkte oder indirekte) Oberrolle von **RoleClass**$_j$. Wenn nun **RoleClass**$_i$ die direkte Oberrolle von **RoleClass**$_j$ ist, dann kann die `create-RoleClass`$_j$-Operation nur aufgerufen werden, wenn das **RoleClass**$_i$-Objekt noch existiert. Falls dagegen **RoleClass**$_i$ eine indirekte Oberrolle darstellt, dann muß es eine Unterrolle **RoleClass**$_k$ von **RoleClass**$_i$ geben, die die direkte Oberrolle von **RoleClass**$_j$ ist. Aufgrund der Semantik der `delete`-Operation (vgl. S. 101) ist sichergestellt, daß ein **RoleClass**$_k$-Objekt nur existieren kann, wenn auch alle Oberrollenobjekte noch vorhanden sind; somit muß auch in diesem Fall ein **RoleClass**$_i$-Objekt existieren. Da zusätzlich eine 1-zu-1 Kardinalität zwischen einem Rollenobjekt und seinem korrespondierenden Objekt vorliegt (vgl. Abb. 44 (S. 99)), muß auch das korrespondierende **Class**$_i$-Objekt existent sein.

Zusammen mit der Restriktion R3 ist gewährleistet, daß bei der Erzeugung des **RoleClass**$_j$-Objekts die Beziehung des **Class**$_j$-Objekts zu dem **Class**$_i$-Objekt ordnungsgemäß aufgebaut wird. Damit können alle Operationen aus **Class**$_j$, die diese Beziehung nutzen, auch korrekt ausgeführt werden.

Für die Konstruktoren der Wurzelklasse **RoleClass**$_1$ sind keine Strukturänderungen vorzunehmen, da die Restriktionen R1 und R2 ausschließen, daß die korrespondierende Klasse **Class**$_1$ Assoziationen zu anderen korrespondierenden Klassen besitzt.

Eine Beispielimplementierung der Klassen **Mitarbeiter** und **Angestellter** aus Abb. 51 (S. 120) ist in Prg. 13 (S. 122) und Prg. 14 (S. 123) zu sehen. Prg. 15 (S. 123) bis Prg. 17 (S. 125) enthalten die Quelltexte der generierten Rollenklassen.

```
 1: package roleModel.singleInheritance;
 2:
 3: class Mitarbeiter {
 4:   private String name;
 5:   private String vorname;
 6:   private double gehalt;
 7:
 8:   public Mitarbeiter ( String name, String vorname,
 9:                        double gehalt ) {
10:     this.name = name;
11:     this.vorname = vorname;
12:     this.gehalt = gehalt;
13:   } // Konstruktor
14:
15:   public void erhoeheGehalt ( double prozent )
16:        throws WrongParameter {
17:     if ( prozent <= 0.0 )
18:        throw new WrongParameter ( "Negativer Prozentwert!!!" );
19:     gehalt = gehalt * ( 1 + prozent/100 );
20:   } // erhoeheGehalt
21:
22:   public void ausgebenDaten () {
23:     System.out.println ( "Name: " + vorname
24:                          + " " + name + " Gehalt=" + gehalt );
25:   } // ausgebenDaten
26:
27: } // Mitarbeiter
```

Prg. 13 Die Klasse **Mitarbeiter**

```
 1: package roleModel.singleInheritance;
 2:
 3: class Angestellter {
 4:   private Mitarbeiter mitarbeiter;
 5:   private String svnummer;
 6:
 7:   public Angestellter ( Mitarbeiter m, String svnr ) {
 8:     mitarbeiter = m;
 9:     svnummer = svnr;
10:   }
11:
12:   public void ausgebenDaten () {
13:     mitarbeiter.ausgebenDaten();
14:     System.out.println
15:       ( "Sozialversicherungsnummer: " + svnummer );
16:   } // ausgebenDaten
17:
18: } // Angestellter
```

Prg. 14 Die Klasse **Angestellter**

```
 1: package roleModel.singleInheritance;
 2:
 3: import scanner.*;
 4:
 5: public class RoleMitarbeiter {
 6:
 7:   Mitarbeiter mitarbeiter = null;
 8:
 9:   RoleAngestellter roleAngestellter = null;
10:
11:   public RoleMitarbeiter ( String name, String vorname,
12:            double gehalt ) {
13:     mitarbeiter = new Mitarbeiter ( name,vorname,gehalt );
14:   }
15:
16:   public void erhoeheGehalt ( double prozent )
17:          throws WrongParameter, RoleDeleted {
18:     if ( mitarbeiter==null )
19:        throw new RoleDeleted ( "Mitarbeiter" );
20:     mitarbeiter.erhoeheGehalt ( prozent );
21:   }
```

Prg. 15 Die Rolle **RoleMitarbeiter** – Teil 1

```
22:
23:    public void ausgebenDaten ()
24:           throws RoleDeleted {
25:      if ( mitarbeiter==null )
26:         throw new RoleDeleted ( "Mitarbeiter" );
27:      mitarbeiter.ausgebenDaten ();
28:    }
29:
30:    public RoleAngestellter createRoleAngestellter
31:             ( String svnr )
32:           throws SubroleExists,RoleDeleted {
33:      if ( mitarbeiter==null )
34:         throw new RoleDeleted ( "Mitarbeiter" );
35:      if ( roleAngestellter!=null )
36:         throw new SubroleExists ( "Angestellter" );
37:      roleAngestellter = new RoleAngestellter ( svnr,this,
38:         mitarbeiter );
39:      return roleAngestellter;
40:    }
41:
42:    public boolean hasSubroleAngestellter()
43:           throws RoleDeleted {
44:      if ( mitarbeiter==null )
45:         throw new RoleDeleted ( "Mitarbeiter" );
46:      return roleAngestellter!=null &&
47:             roleAngestellter.angestellter!=null;
48:    }
49:
50:    public boolean isDeleted () {
51:      return mitarbeiter == null;
52:    }
53:
54:    public void delete () throws RoleDeleted {
55:      if ( mitarbeiter==null )
56:         throw new RoleDeleted ( "Mitarbeiter" );
57:      mitarbeiter = null;
58:      if ( roleAngestellter != null )
59:         roleAngestellter.delete();
60:    }
61:
62:  }
```

Prg. 16 | Die Rolle **RoleMitarbeiter** – Teil 2

```
 1: package roleModel.singleInheritance;
 2:
 3: import scanner.*;
 4:
 5: public class RoleAngestellter {
 6:
 7:   Angestellter angestellter = null;
 8:
 9:   RoleMitarbeiter roleMitarbeiter = null;
10:
11:   Mitarbeiter mitarbeiter = null;
12:
13:
14:   RoleAngestellter ( String svnr,
15:       RoleMitarbeiter roleMitarbeiter, Mitarbeiter mitarbeiter ) {
16:     angestellter = new Angestellter ( mitarbeiter,svnr );
17:     this.roleMitarbeiter = roleMitarbeiter;
18:     this.mitarbeiter = mitarbeiter;
19:   }
20:
21:   public void ausgebenDaten ()
22:          throws RoleDeleted {
23:     if ( angestellter==null )
24:        throw new RoleDeleted ( "Angestellter" );
25:     angestellter.ausgebenDaten ();
26:   }
27:
28:   public void erhoeheGehalt ( double prozent )
29:          throws WrongParameter, RoleDeleted {
30:     if ( mitarbeiter==null )
31:        throw new RoleDeleted ( "Mitarbeiter" );
32:     mitarbeiter.erhoeheGehalt ( prozent );
33:   }
34:
35:   public boolean isDeleted () {
36:     return angestellter == null;
37:   }
38:
39:   public void delete () throws RoleDeleted {
40:     if ( angestellter==null )
41:        throw new RoleDeleted ( "Angestellter" );
42:     angestellter = null;
43:     mitarbeiter = null;
44:     roleMitarbeiter.roleAngestellter = null;
45:     roleMitarbeiter = null;
46:   }
47:
48: }
```

Prg. 17 Die Rolle **RoleAngestellter**

5.2.6 Anwendung des Vererbungskonzepts bei der Definition der korrespondierenden Klassen

Da die Vererbung ein Standardkonzept der objektorientierten Analyse ist, sollte sie auch für die Modellierung der korrespondierenden Klassen **Class**$_1$ bis **Class**$_n$ zur Verfügung stehen. Strukturell sind hier zwei Fälle zu unterscheiden:

- Alle (direkten und indirekten) Oberklassen einer korrespondierenden Klasse gehören *nicht* zur Menge der korrespondierenden Klassen des Rollenmodells.
- Unter den Oberklassen befinden sich auch (andere) korrespondierende Klassen des Rollenmodells.

Die Menge der Oberklassen einer korrespondierenden Klasse enthält selbst keine korrespondierende Klasse

Abb. 52 (S. 127) zeigt ein Rollenmodell, in dem die beiden korrespondierenden Klassen **AB** und **BB** jeweils eine Oberklasse besitzen (**A** bzw. **B**). Bei der Erzeugung der Operationen der Rollenklassen ist zu berücksichtigen, daß eine korrespondierende Klasse **Class**$_i$ ($i \neq 1$) jetzt in zwei Hierarchien enthalten sein kann:

- Die Baumstruktur, die zur Definition der Rollenhierarchie dient.
- Die Vererbungshierarchie, die (möglicherweise) für **Class**$_i$ definiert ist.

Für das Beispiel aus Abb. 52 (S. 127) besitzt die korrespondierende Klasse **BB** diese Eigenschaft: zum einem hat sie die Oberklasse **B**, zum anderen besitzt sie in der Baumstruktur der Rollenhierarchie als direkten Vorgänger die Klasse **AB**. Die Operation `mB1` aus **BB** redefiniert die entsprechende Operation aus **B** und verdeckt zusätzlich die Operation `mB1` aus **AB** in Bezug auf die Rollenhierarchie, die ihrerseits wieder die Operation `mB1` aus **A** redefiniert. Beim Erzeugungsprozeß der Operationen einer Rollenklasse **RoleClass**$_i$ ist folgende Verarbeitungsreihenfolge einzuhalten:

- Zunächst werden alle Operationen aus der korrespondierenden Klasse **Class**$_i$ übernommen, die öffentlich sichtbar sind.
- Im zweiten Schritt sind alle Oberklassen von **Class**$_i$ nach weiteren öffentlichen Operationen zu durchsuchen. Sie werden in **RoleClass**$_i$ übernommen, sofern eine entsprechende Operation nicht bereits im ersten Schritt erzeugt wurde.
- Abschließend findet die Verarbeitung der Oberrollen von **RoleClass**$_i$ bis zur Wurzelrolle **RoleClass**$_1$ statt. Dabei wird für eine Rolle **RoleClass**$_j$ zunächst immer erst ihre korrespondierende Klasse **Class**$_j$ betrachtet, gefolgt von ihren Oberklassen. Auch hier gilt wieder, daß eine Operation nur dann nach **RoleClass**$_i$ übernommen wird, wenn nicht eine entsprechende Operation bereits in den vorherigen Schritten erzeugt wurde.

In der Rollenklasse **RoleAB** aus Abb. 52 (S. 127) wird zur Realisierung der Operation `mB1` auf die entsprechende Operation aus **AB** zugegriffen (vgl. Zeile 29 in Prg. 22 (S. 131)). Die `mB1`-Operation in **RoleBB** verwendet dagegen die `mB1`-Operation aus **BB**, da die korrespondierende Klasse und ihre Oberklassen vor den Oberrollen verarbeitet werden (vgl. Zeile 20 in Prg. 24 (S. 133)). Aus dem gleichen Grund benutzt die Operation `mA2` aus der Rollenklasse **RoleBB** die entsprechende Operation aus der Oberklasse **B** von **BB** und nicht die `mA2`-Operation aus

der korrespondierenden Klasse der Oberrolle **RoleAB** (vgl. Zeile 41 in Prg. 24 (S. 133)).

Alle anderen Bestandteile der Rollenklassen (Konstruktoren, `createRole`-, `hasSubrole`-, `isDeleted`- und `delete`-Operationen) sind von der Integration des Vererbungskonzepts für die korrespondierenden Klassen nicht betroffen.

Die korrespondierenden Klassen mit ihren Oberklassen für das über Abb. 52 definierte Rollenmodell sind in Prg. 18 (S. 128) bis Prg. 21 (S. 130) dargestellt. Die generierten Programme für die beiden Rollenklassen **RoleAA** und **RoleAB** findet man in Prg. 22 (S. 131) bis Prg. 25 (S. 134).

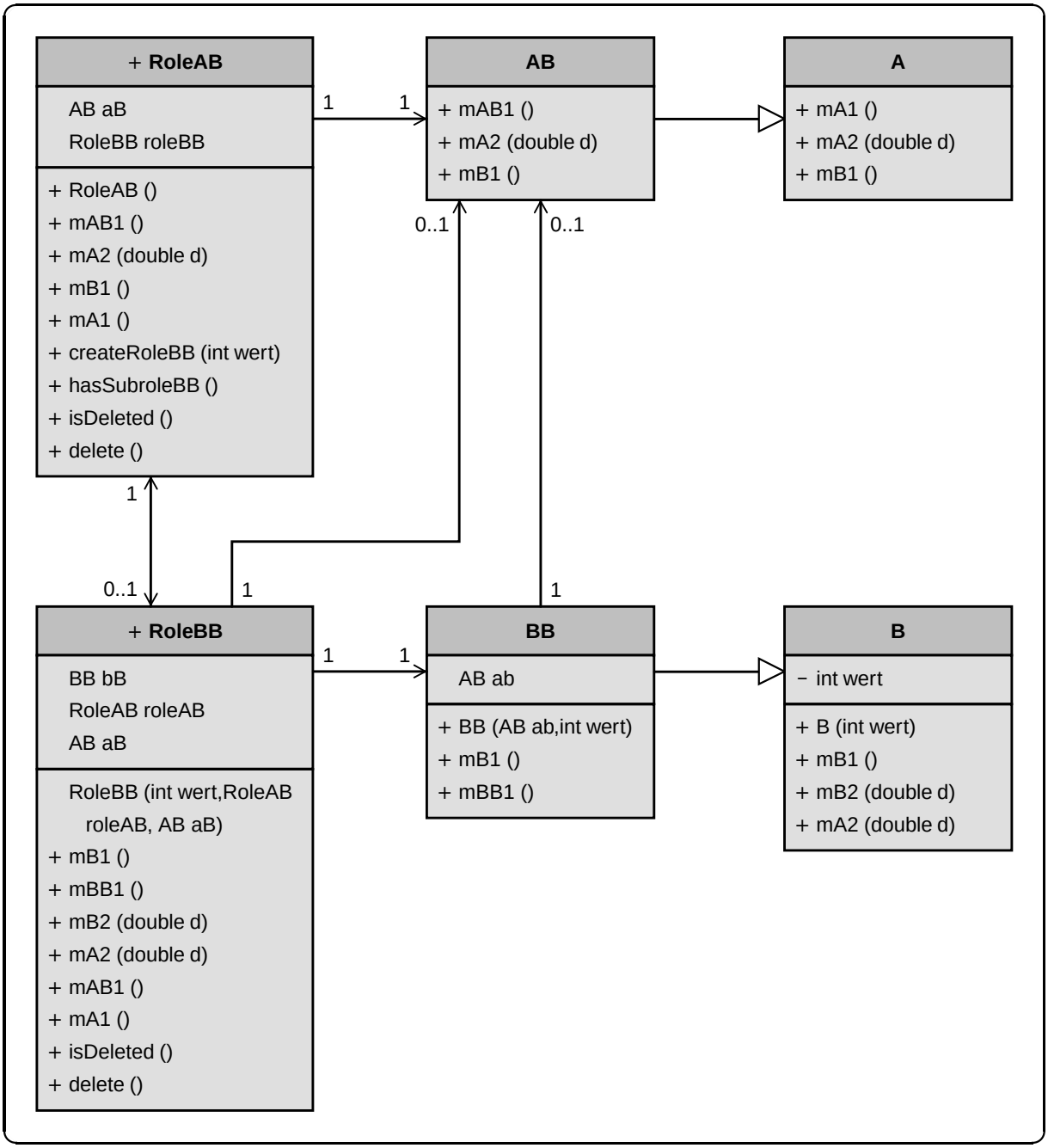


Abb. 52 Das Vererbungskonzept bei korrespondierenden Klassen

```
 1: package roleModel.singleInheritance;
 2:
 3: class A {
 4:
 5:   public void mA1 () {
 6:     System.out.println ( "mA1() aus A" );
 7:   } // mA1
 8:
 9:   public int mA2 ( double d ) {
10:     System.out.println ( "mA2(d) aus A: d=" + d );
11:     return -1;
12:   } // mA2
13:
14:   public void mB1() {
15:     System.out.println ( "mB1() aus A" );
16:   } // mB1
17:
18: } // A
```

Prg. 18 Die Klasse **A**

```
 1: package roleModel.singleInheritance;
 2:
 3: class AB extends A {
 4:
 5:   public void mAB1 () {
 6:     System.out.println ( "mAB1() aus AB" );
 7:   } // mAB1
 8:
 9:   public int mA2 ( double d ) {
10:     System.out.println ( "mA2(d) aus AB: d=" + d );
11:     return -8;
12:   } // mA2
13:
14:   public void mB1() {
15:     System.out.println ( "mB1() aus AB");
16:     super.mB1();
17:   } // mB1
18:
19: } // AB
```

Prg. 19 Die Klasse **AB**

```
 1: package roleModel.singleInheritance;
 2:
 3: class B {
 4:   private int wert;
 5:
 6:   public B ( int wert ) {
 7:     this.wert = wert;
 8:   }
 9:
10:   public void mB1 () {
11:     System.out.println ( "mB1() aus B: wert=" + wert );
12:   } // mB1
13:
14:   public int mB2 ( double d ) {
15:     System.out.println ( "mB2(d) aus B: d=" + d );
16:     return -20;
17:   } // mB2
18:
19:   public int mA2 ( double d ) {
20:     System.out.println ( "mA2(d) aus B: d=" + d );
21:     return -25;
22:   } // mA2
23:
24: } // B
```

Prg. 20 Die Klasse **B**

```
 1: package roleModel.singleInheritance;
 2:
 3: class BB extends B {
 4:   AB ab = null;
 5:
 6:   public BB ( AB ab, int wert ) {
 7:     super ( wert );
 8:     this.ab = ab;
 9:   }
10:
11:   public void mB1 () {
12:     System.out.println ( "mB1() aus BB" );
13:     ab.mB1();
14:     super.mB1();
15:   } // mB1
16:
17:   public void mBB1 () {
18:     System.out.println ( "mBB1() aus BB" );
19:   } // mBB1
20:
21: } // BB
```

Prg. 21 Die Klasse **BB**

```
 1: public class RoleAB {
 2:
 3:   AB aB = null;
 4:
 5:   RoleBB roleBB = null;
 6:
 7:   public RoleAB () {
 8:     aB = new AB();
 9:   }
10:
11:   public void mAB1 ()
12:          throws RoleDeleted {
13:     if ( aB==null )
14:        throw new RoleDeleted ( "AB" );
15:     aB.mAB1 ();
16:   }
17:
18:   public int mA2 ( double d )
19:          throws RoleDeleted {
20:     if ( aB==null )
21:        throw new RoleDeleted ( "AB" );
22:     return aB.mA2 ( d );
23:   }
24:
25:   public void mB1 ()
26:          throws RoleDeleted {
27:     if ( aB==null )
28:        throw new RoleDeleted ( "AB" );
29:     aB.mB1 ();
30:   }
31:
32:   public void mA1 ()
33:          throws RoleDeleted {
34:     if ( aB==null )
35:        throw new RoleDeleted ( "AB" );
36:     aB.mA1 ();
37:   }
38:
39:   public RoleBB createRoleBB ( int wert )
40:          throws SubroleExists,RoleDeleted {
41:     if ( aB==null )
42:        throw new RoleDeleted ( "AB" );
43:     if ( roleBB!=null )
44:        throw new SubroleExists ( "BB" );
45:     roleBB = new RoleBB ( wert,this,aB );
46:     return roleBB;
47:   }
```

Prg. 22 Die Rolle **RoleAB** – Teil 1

```
48:
49:    public boolean hasSubroleBB()
50:           throws RoleDeleted {
51:      if ( aB==null )
52:         throw new RoleDeleted ( "AB" );
53:      return roleBB!=null &&
54:             roleBB.bB!=null;
55:    }
56:
57:    public boolean isDeleted () {
58:      return aB == null;
59:    }
60:
61:    public void delete () throws RoleDeleted {
62:      if ( aB==null )
63:         throw new RoleDeleted ( "AB" );
64:      aB = null;
65:      if ( roleBB != null )
66:         roleBB.delete();
67:    }
68:
69:  }
```

Prg. 23 Die Rolle **RoleAB** – Teil 2

```
 1: public class RoleBB {
 2:
 3:   BB bB = null;
 4:
 5:   RoleAB roleAB = null;
 6:
 7:   AB aB = null;
 8:
 9:
10:   RoleBB ( int wert, RoleAB roleAB, AB aB ) {
11:     bB = new BB ( aB,wert );
12:     this.roleAB = roleAB;
13:     this.aB = aB;
14:   }
15:
16:   public void mB1 ()
17:          throws RoleDeleted {
18:     if ( bB==null )
19:        throw new RoleDeleted ( "BB" );
20:     bB.mB1 ();
21:   }
22:
23:   public void mBB1 ()
24:          throws RoleDeleted {
25:     if ( bB==null )
26:        throw new RoleDeleted ( "BB" );
27:     bB.mBB1 ();
28:   }
29:
30:   public int mB2 ( double d )
31:          throws RoleDeleted {
32:     if ( bB==null )
33:        throw new RoleDeleted ( "BB" );
34:     return bB.mB2 ( d );
35:   }
36:
37:   public int mA2 ( double d )
38:          throws RoleDeleted {
39:     if ( bB==null )
40:        throw new RoleDeleted ( "BB" );
41:     return bB.mA2 ( d );
42:   }
```

Prg. 24 Die Rolle **RoleBB** – Teil 1

```
43:
44:    public void mAB1 ()
45:           throws RoleDeleted {
46:      if ( aB==null )
47:         throw new RoleDeleted ( "AB" );
48:      aB.mAB1 ();
49:    }
50:
51:    public void mA1 ()
52:           throws RoleDeleted {
53:      if ( aB==null )
54:         throw new RoleDeleted ( "AB" );
55:      aB.mA1 ();
56:    }
57:
58:    public boolean isDeleted () {
59:      return bB == null;
60:    }
61:
62:    public void delete () throws RoleDeleted {
63:      if ( bB==null )
64:         throw new RoleDeleted ( "BB" );
65:      bB = null;
66:      aB = null;
67:      roleAB.roleBB = null;
68:      roleAB = null;
69:    }
70:
71:  }
```

Prg. 25 Die Rolle **RoleBB** – Teil 2

Die Menge der Oberklassen einer korrespondierenden Klasse enthält selbst wieder mindestens eine korrespondierende Klasse

In Abb. 53 (S. 135) wurde die Rollenhierarchie aus Abb. 9 (S. 16) um die beiden Rollen **RoleTutor** und **RolePromotionsstudent** ergänzt. Ein wissenschaftlicher Mitarbeiter kann jetzt die Rolle eines Promotionsstudenten annehmen. Wenn die Funktionalitäten der Rollen Student und Promotionsstudent identisch wären, würde es sich auf den ersten Blick anbieten, **RoleStudent** nicht nur als Unterrolle von **RolePerson** sondern auch als Unterrolle von **RoleWissenschaftlicherMitarbeiter** zu modellieren. Dies wäre dann ein Beispiel für die Anwendung der Mehrfachvererbung innerhalb eines Rollenmodells, da **RoleStudent** jetzt zwei direkte Oberrollen besitzt. Dagegen sprechen jedoch zwei Aspekte: zum einen würde man damit verlangen, daß jeder Student auch wissenschaftlicher Mitarbeiter ist, da eine Unterrolle nur erzeugt werden darf, wenn das entsprechende Objekt bereits alle direkten Oberrollen innehat; zum anderen könnte ein wissenschaftlicher Mitarbeiter in seiner Rolle als Student eine Tutorenrolle übernehmen, was üblicherweise nicht erlaubt ist.

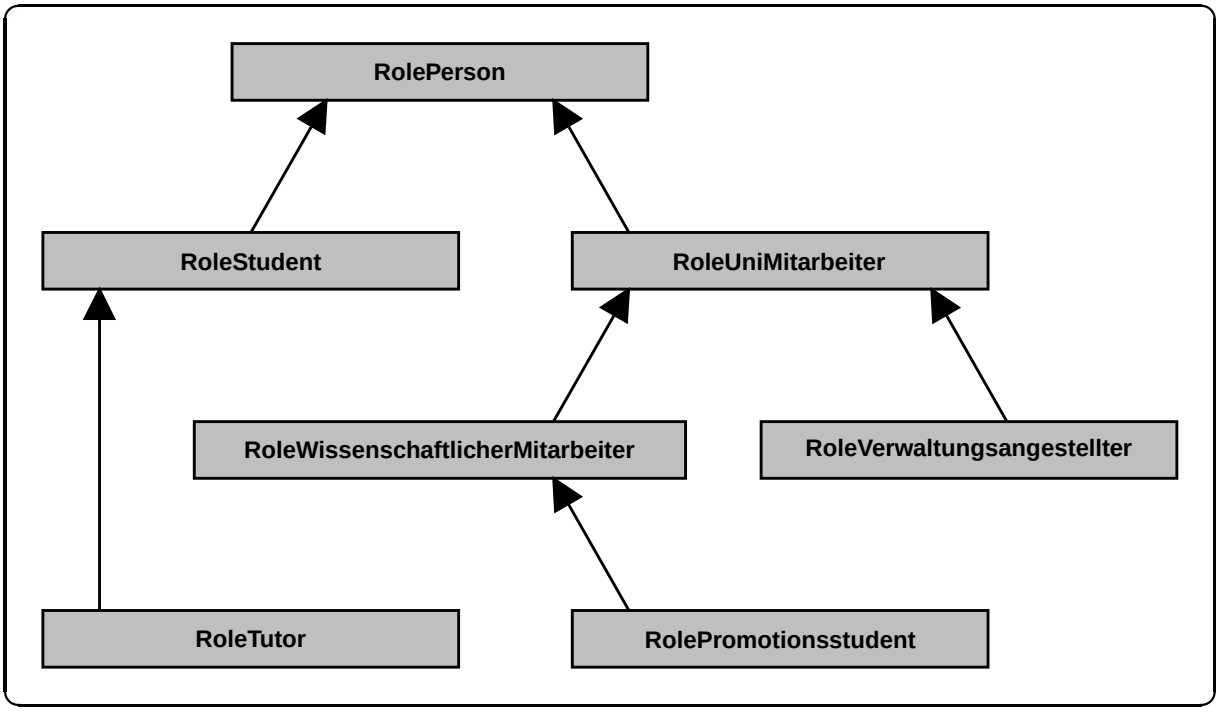


Abb. 53 Modellierung von Studenten- und Promotionsstudentenrollen

Da innerhalb eines Rollenmodells die Rollenklassen sowohl bezüglich ihrer Funktionalität als auch bezüglich der Namensgebung über die korrespondierenden Klassen definiert sind, kann eine korrespondierende Klasse nicht zur Erzeugung von unterschiedlichen Rollenklassen benutzt werden. Um aber unnötige Redundanz bei der Implementierung der korrespondierenden Klassen des Rollenmodells zu vermeiden, ist es sinnvoll, zwischen den korrespondierenden Klassen **Student** und **Promotionsstudent** eine Vererbungsbeziehung zu verwenden.

Vererbungsbeziehungen sollten jedoch nicht zwischen beliebigen korrespondierenden Klassen definiert werden, damit nicht die Gefahr einer redundanten Datenhaltung entsteht. Abb. 54 zeigt die Struktur eines einfachen Rollenmodells, wobei zwischen den korrespondierenden Klassen **D** und **B** eine Vererbungsbeziehung besteht, wie in der Umsetzung des Rollenmodells in Abb. 55 (S. 136) zu erkennen ist.

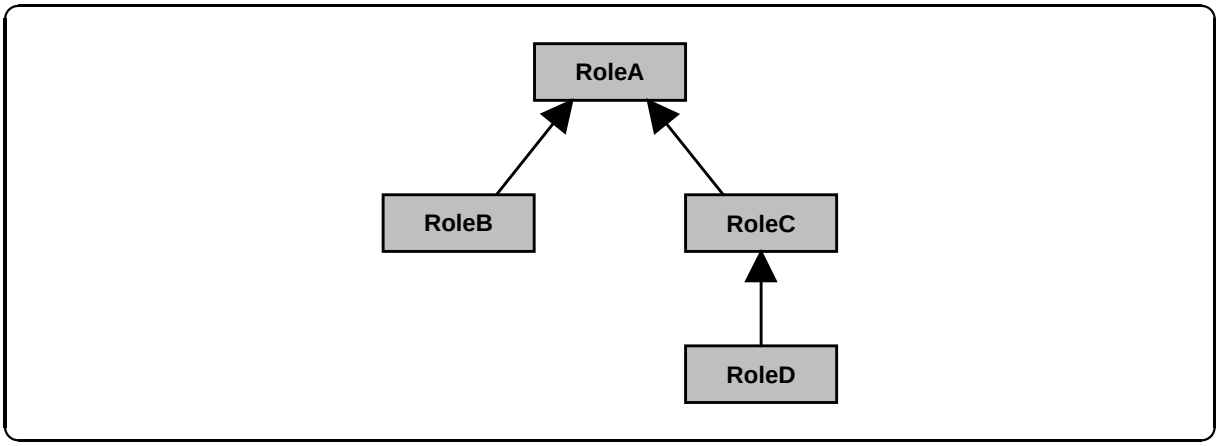


Abb. 54 Vererbung zwischen korrespondierenden Klassen

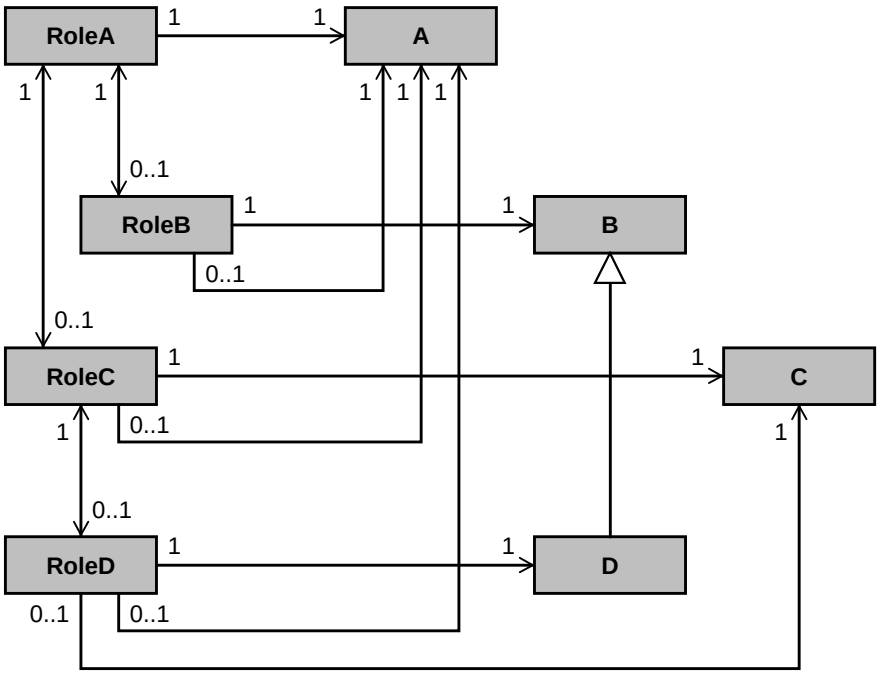


Abb. 55 Umsetzung des Rollenmodells aus Abb. 54 (S. 135)

Wenn ein Objekt des Typs **RoleA** gleichzeitig die Rollen **RoleB** und **RoleD** annehmen kann, dann existieren zwei korrespondierende Objekte, die potentiell redundante Daten besitzen können, da das korrespondierende Objekt zu **RoleD** aufgrund der Vererbungsbeziehung alle Attribute besitzt, die für das korrespondierende Objekt zu **RoleB** definiert sind. Man sollte sich daher auf der Analyseebene überlegen, ob nicht eine **exor**-Restriktion zwischen **RoleB** und **RoleD** sinnvoll ist.[8] Die Gefahr der Datenredundanz liegt dagegen grundsätzlich vor, wenn eine Vererbungsbeziehung zwischen zwei korrespondierenden Klassen existiert, deren Rollenklassen in einer direkten oder indirekten Rollenbeziehung stehen. In Abb. 56 (S. 137) liegt eine Vererbungsbeziehung zwischen den Klassen **D** und **A** vor. Durch das vorgegebene Rollenmodell (vgl. Abb. 54 (S. 135)) existieren immer dann Objekte der Klassen **D** und **A**, wenn eine Rolle des Typs **RoleD** vorhanden ist. Bei der Umsetzung eines Rollenmodells sollte daher sorgfältig überdacht werden, ob ein derartiger Fall wirklich gewünscht ist.

Abschließend sei zum Thema Vererbung bemerkt, daß zwei korrespondierende Klassen natürlich eine gemeinsame Oberklasse besitzen dürfen. Für das Beispiel einer Universitätsverwaltung aus Abb. 53 (S. 135) wäre es denkbar, daß eine direkte Vererbungsbeziehung zwischen den Klassen **Student** und **Promotionsstudent** aus Analysesicht nicht geeignet ist, weil jede Klasse mindestens eine Eigenschaft besitzt, die in der jeweils anderen Klasse nicht benötigt wird. Dann bietet es sich an, eine gemeinsame Oberklasse für diese beiden korrespondierenden Klassen zu verwenden, die die Gemeinsamkeiten beider Studentenklassen enthält und typischerweise als abstrakte Klasse modelliert und implementiert wird. Auch hier gilt wieder, daß eine derartige Vererbungsbeziehung in der Regel nur sinnvoll sein dürfte, wenn die zugehörigen Rollenklassen in keiner direkten oder indirekten Rollenbeziehung stehen.

[8] Die Spezifikation von **exor**-Restriktionen innerhalb des Rollenmodells wird in Abschnitt 5.6.2 (S. 174) behandelt.

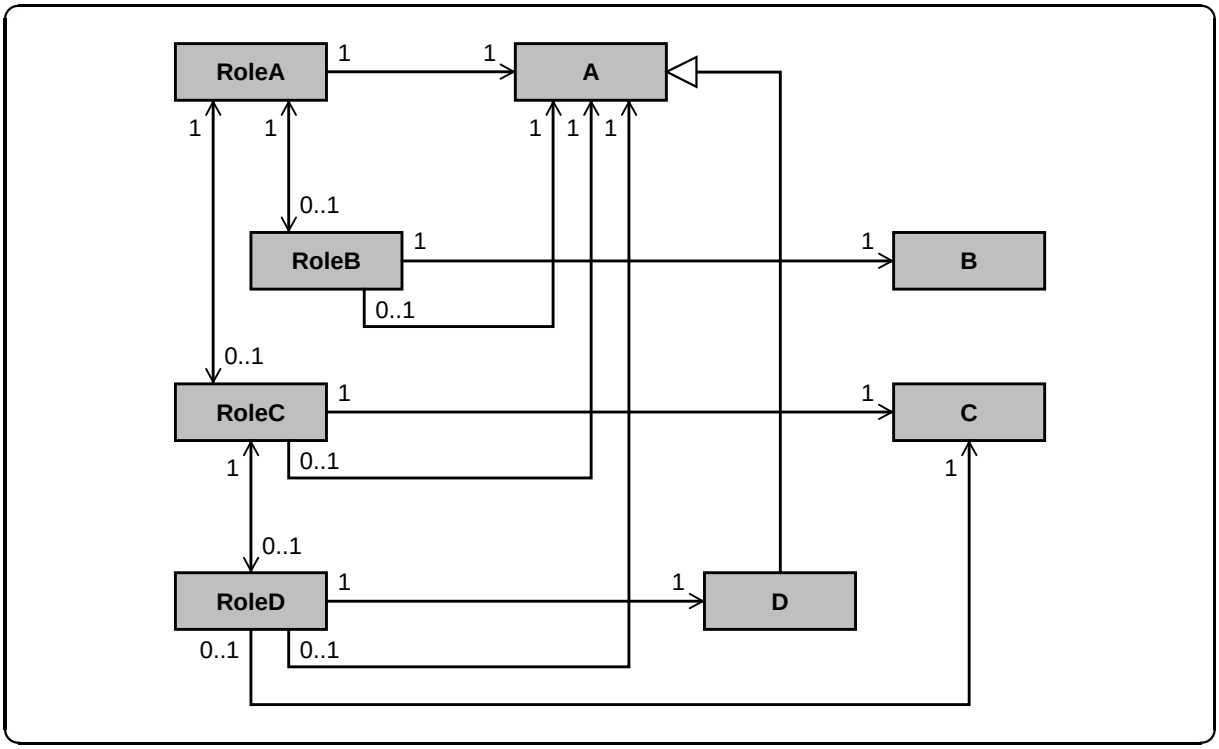


Abb. 56 Alternative Umsetzung des Rollenmodells aus Abb. 54 (S. 135)

Abb. 57 zeigt die entstehende Struktur, wenn die beiden korrespondierenden Klassen **D** und **B** des Rollenmodells aus Abb. 54 (S. 135) eine gemeinsame abstrakte Oberklasse **X** besitzen.

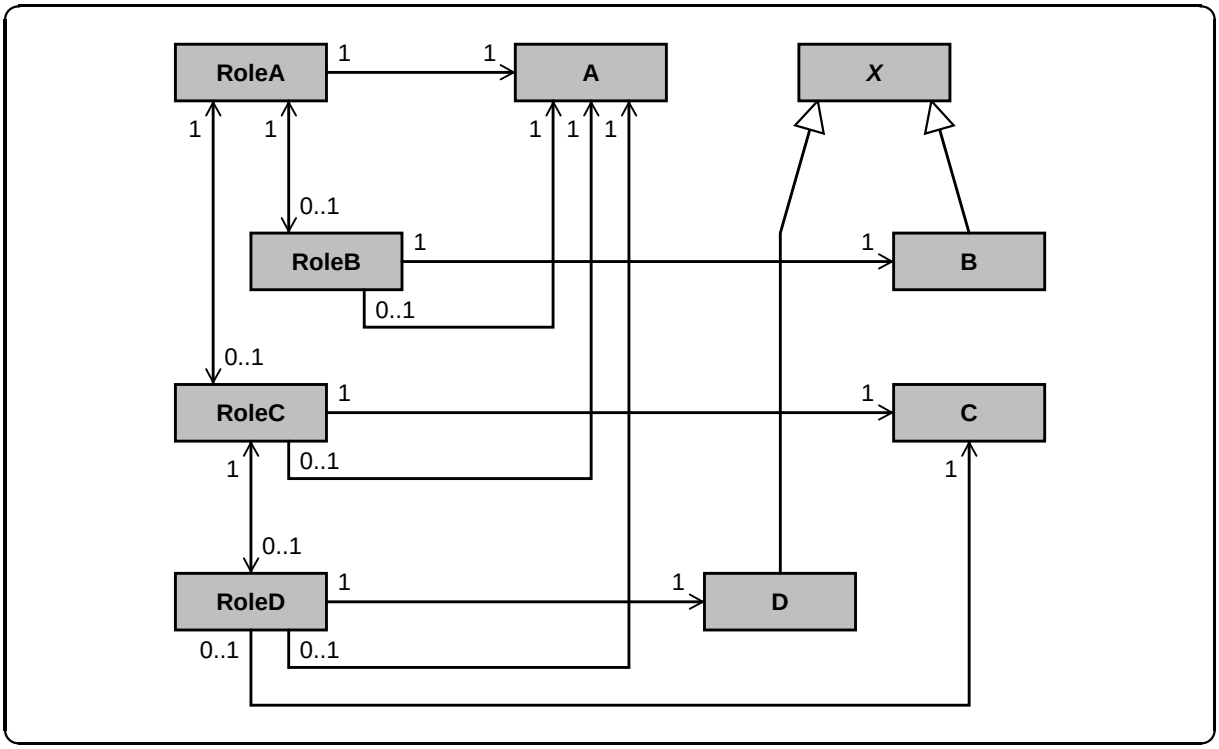


Abb. 57 Verwendung einer abstrakten Klasse für das Rollenmodell aus Abb. 54 (S. 135)

Für die Generierung der Rollenmodelle hat die Hinzunahme von Vererbungsbeziehungen keine weiteren Auswirkungen. Alle notwendigen Aspekte sind bereits durch die auf S. 126 beschriebenen Funktionalitätserweiterungen abgedeckt, die für die Umsetzung der Vererbungsbeziehungen korrespondierender Klassen zu nicht-korrespondierenden Klassen erforderlich waren.

5.3 Rollenmodelle mit Unterrollenkardinalitäten größer als 1

5.3.1 Analysemodell

Bisher wurde vorausgesetzt, daß eine Unterrolle höchstens einmal angenommen werden kann. Dieser Modellierungsansatz ist in vielen Anwendungsfällen sicherlich zu restriktiv. Betrachtet man das Beispiel einer Universitätsverwaltung aus Abb. 9 (S. 16), so liefert eine Analyse des Universitätsalltags, daß ein Student durchaus *mehrere* Tutorentätigkeiten übernehmen kann, die jeweils eigenständige Instanzen einer Tutorenrolle darstellen.

Auf der Analyseebene sind nur geringfügige Änderungen vorzunehmen: die Kardinalität 0..1 ist durch die gewünschte Kardinalität zu ersetzen. Abb. 58 zeigt auf der linken Seite die Definition eines Rollenmodells, bei dem ein Student beliebig viele Tutorentätigkeiten ausüben kann, während auf der rechten Seite die Anzahl auf maximal 4 beschränkt ist.

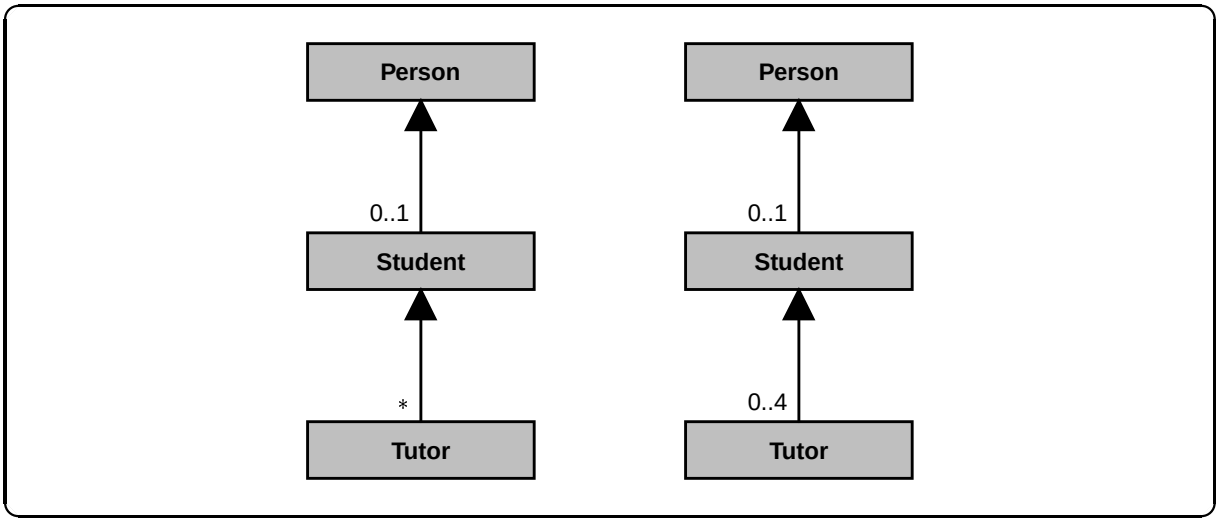


Abb. 58 Die Baumstruktur zur Definition eines Rollenmodells mit Kardinalitäten > 1

In der Folge wird angenommen, daß die Minimalkardinalität 0 und die Maximalkardinalität eine feste Zahl oder unendlich (d.h. $*$) ist. Prinzipiell wäre natürlich auch eine andere Definition der Kardinalitäten denkbar. Es erscheint bei Rollenmodellen aber wenig sinnvoll, z.B. eine untere Grenze k größer als 0 vorzugeben. Dann müßten mit der Erzeugung der Oberrolle automatisch auch (mindestens) k Instanzen der zugehörigen Unterrolle erzeugt werden. Außerdem wäre mit dem Löschen der k-ten Unterrolleninstanz auch die entsprechende Oberrolleninstanz zu löschen und damit natürlich auch die anderen $k-1$ Unterrolleninstanzen, da die Existenz der Oberrolleninstanz fest an das Vorhandensein von mindestens k Unterrolleninstanzen gekoppelt ist. Dies würde auch der Anforderung AR5 (vgl. S. 16) widersprechen, daß Rollen unabhängig voneinander angenommen und wieder abgelegt werden können, sofern keine Restriktionen definiert wurden, die gerade dieses verhindern sollen.

Bei den Operationen zur Verwaltung des Rollenmodells sind die folgenden Ergänzungen bzw. Änderungen vorzunehmen:

- Erzeugung einer Unterrolle:
  Falls eine feste Grenze für die Anzahl der Unterrollen vorgegeben wurde, hat eine `createRoleClass`$_k$-Operation diese zu überprüfen. Ist die Grenze bereits erreicht, wird die Operation mit einer `AllSubrolesExist`-Exception beendet.
- Test, ob bereits alle Unterrollen existieren:
  Die `hasSubroleClass`$_k$-Operation wird durch eine `hasAllSubrolesClass`$_k$-Operation ersetzt[9].
- Löschen einer Rolle:
  Die `delete`-Operation muß beim (rekursiven) Löschen der Unterrollen die Kardinalitäten des jeweiligen Unterrollentyps berücksichtigen.

5.3.2 Entwurfsmodell

Wie ein Vergleich zwischen Abb. 59 und Abb. 44 (S. 99) verdeutlicht, liegt die wesentliche Strukturänderung des Entwurfsmodells in der Verwaltung der Unterrollen eines Rollenobjekts. Statt eines einfachen Attributs wird eine Listenstruktur[10] verwendet.

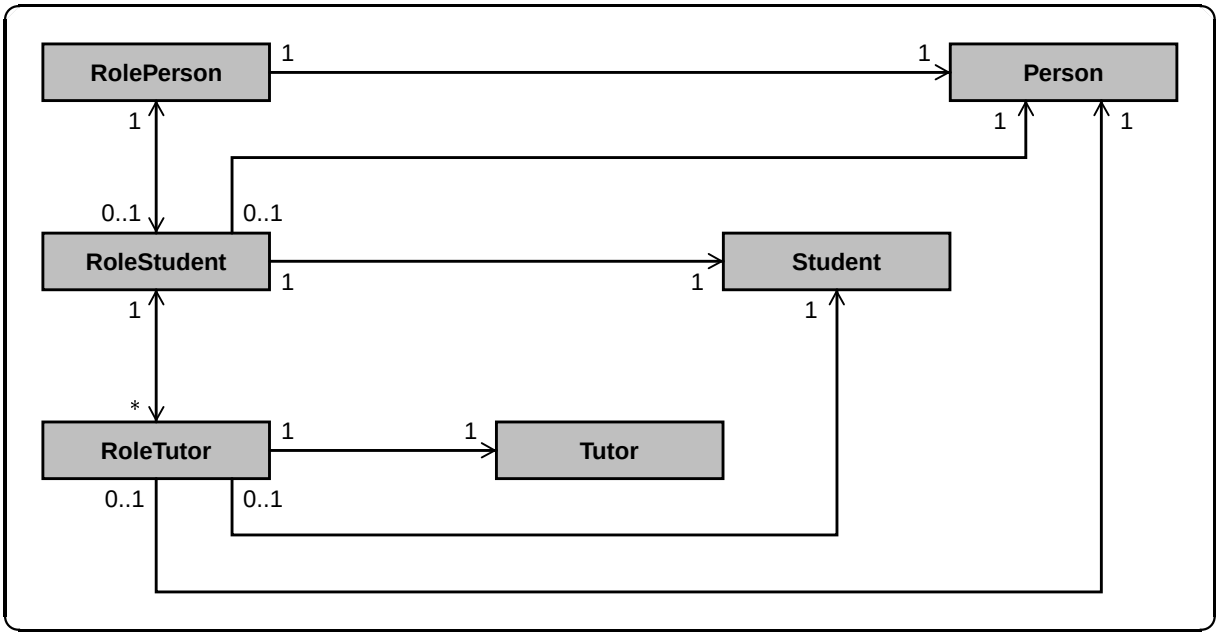


Abb. 59 Klassendiagramm des über Abb. 58 (S. 139) definierten Rollenmodells

Für den Entwurf der Listenstruktur sind mehrere Varianten denkbar, die in der Folge vorgestellt und bewertet werden.

[9] Für den Fall, daß die Obergrenze $*$ definiert wurde, hätte man auf diese Testoperation natürlich auch ganz verzichten können, da ihr Ergebnis immer **`false`** ist.

[10] Für den Fall, daß eine feste Obergrenze für die Anzahl der Unterrollen vorliegt, wäre auch eine Array-Struktur anwendbar. Um nicht zu viele Generierungsvarianten zu erhalten, wird die Listenstruktur gewählt, da diese für die Kardinalität $*$ konzeptuell notwendig ist.

Für die einzelnen Unterrollenklassen spezialisierte Listenstrukturen: L1

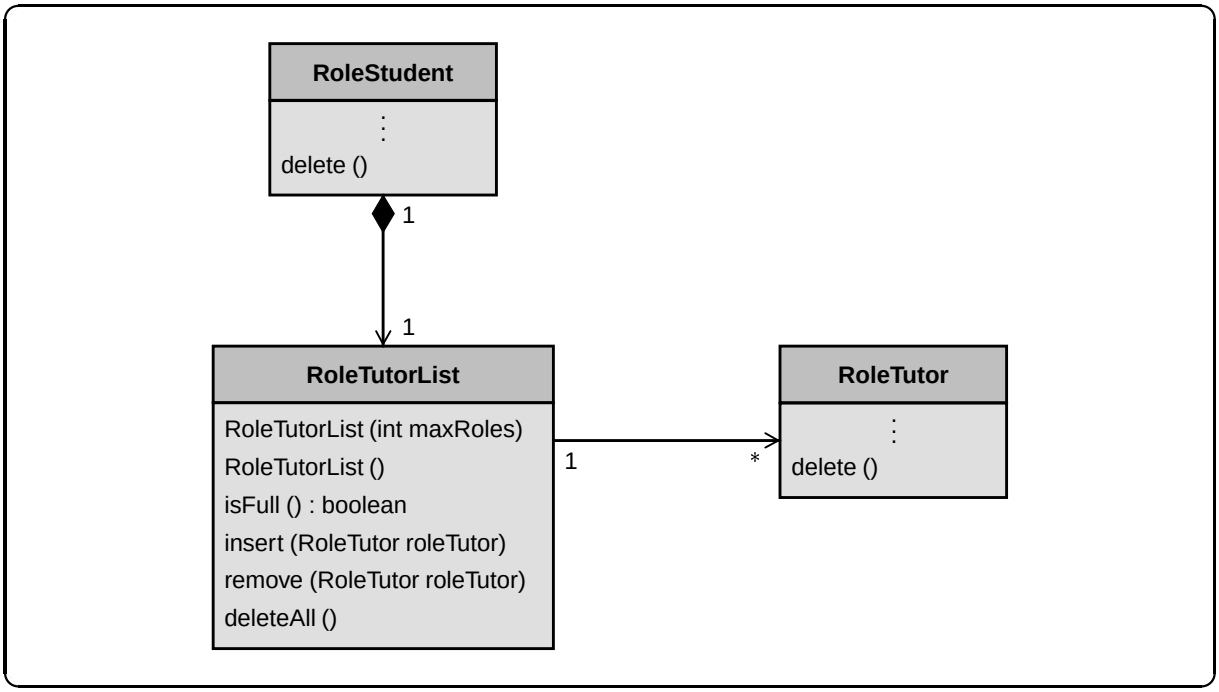


Abb. 60 Erste Variante der Listenstruktur

In dem Entwurf aus Abb. 60 kann bereits der Compiler über die statische Typprüfung kontrollieren, daß die Listenoperationen ausschließlich Objekte des Typs **RoleTutor** betreffen. Der erste Konstruktor ist zu verwenden, wenn für die Kardinalität eine feste Obergrenze vorgegeben ist, der zweite Konstruktor, wenn keine feste Obergrenze spezifiziert wurde. Liegt der erste Fall vor, dann testet die `insert`-Operation, ob die Liste bereits voll ist und erzeugt gegebenenfalls eine Exception. Die `remove`-Operation entfernt das entsprechende Rollenobjekt aus der Listenstruktur, löscht es aber nicht. Stattdessen wird sie innerhalb der `delete`-Operation für ein **RoleTutor**-Objekt verwendet, um die Beziehung vom **RoleStudent**- zum **RoleTutor**-Objekt abzubauen. Die `deleteAll`-Operation hat die Aufgabe, das (rekursive) Löschen für alle **RoleTutor**-Objekte auszuführen; `deleteAll` wird während der Verarbeitung der `delete`-Operation der Oberrolle, d.h. in diesem Fall **RoleStudent**, aufgerufen.

Die erste Variante hat den Nachteil, daß für jede Ober-/Unterrollenbeziehung mit einer Kardinalität größer als 1 eine eigene Listenklasse zu erzeugen ist, obwohl die eigentliche Listenfunktionalität selbst immer identisch ist.

Kombination aus einer allgemeinen und spezialisierten Listenstrukturen: L2

Bei der zweiten Variante (vgl. Abb. 61 (S. 142)) bildet die Klasse **RoleList** die Grundlage für die Listenstrukturen. Sie realisiert eine Liste, deren Elemente vom Interface-Typ **MultipleRole** sein müssen. Damit ist sichergestellt, daß die später verwendeten Klassen eine `delete` sowie eine `isDeleted`-Operation besitzen, die innerhalb der Implementierung der `deleteAll`-Operation aus **RoleList** verwendet wird. Um jetzt für eine Rollenklasse **RoleX** die zugehörige Listenklasse zu erzeugen, muß zunächst **RoleX** das Interface **MultipleRole** implementieren,

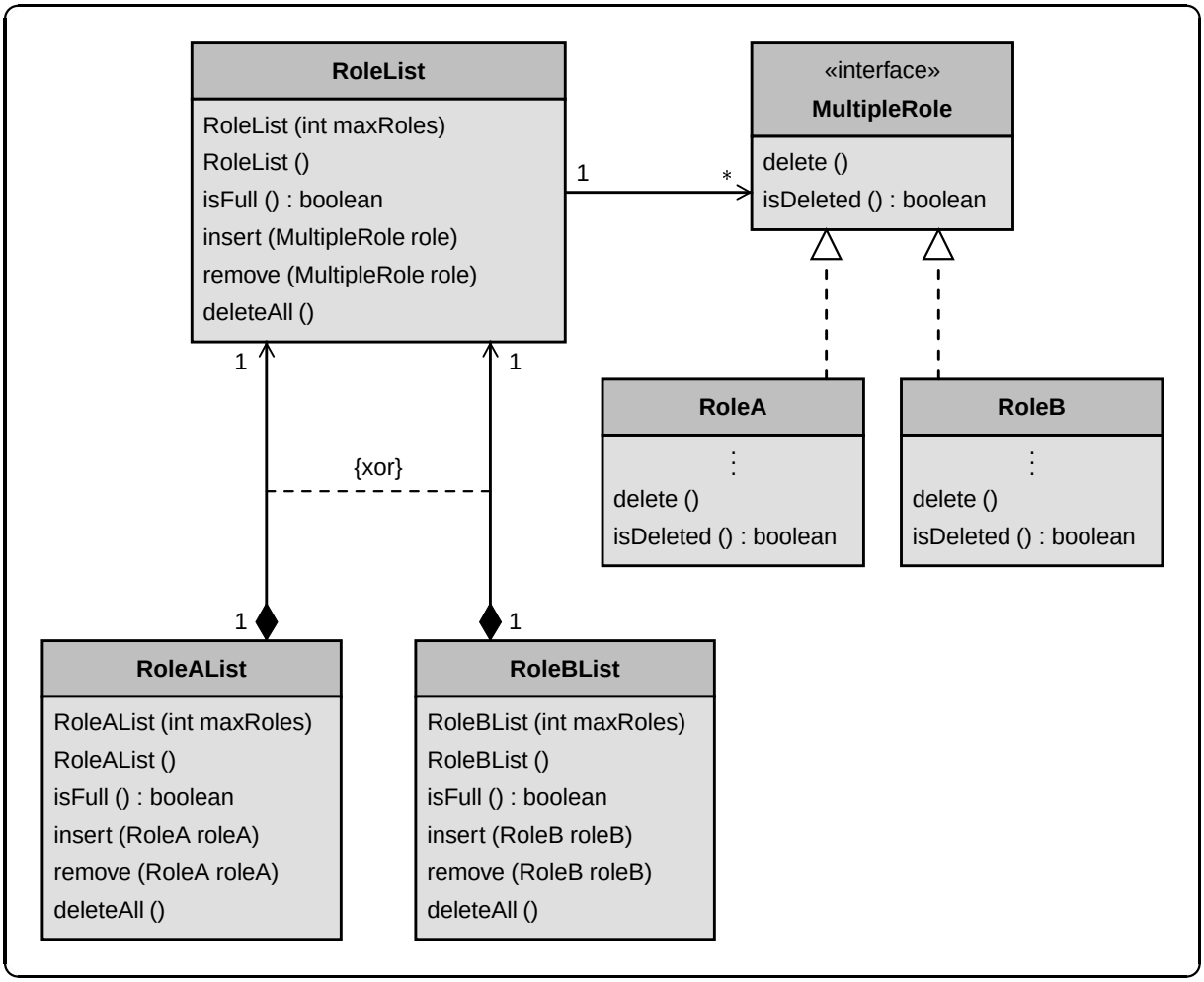


Abb. 61 Zweite Variante der Listenstruktur

was aber aufgrund der bereits vorliegenden Struktur der Rollenklassen keinen zusätzlichen Aufwand darstellt, weil die Rollenklassen standardmäßig über die beiden zu implementierenden Operationen verfügen. Im zweiten Schritt ist eine Listenklasse **RoleXList** zu implementieren, die intern ein Objekt der Klasse **RoleList** verwendet. Die Operationen aus **RoleXList** reichen lediglich die Aufrufparameter an die entsprechenden Operationen aus **RoleX** durch. Die Klasse **RoleXList** wird wie bei der Variante **L1** wieder benutzt, um bereits durch den Compiler überprüfen zu lassen, daß in der Liste **RoleXList** ausschließlich Objekte der Klasse **RoleX** verwaltet werden. In Abb. 61 übernehmen **RoleA** und **RoleB** die Rolle von **RoleX**. Die **exor**-Restriktion ist erforderlich, da ein **RoleList**-Objekt immer eindeutig einem **RoleXList**-Objekt zugeordnet sein muß, d.h. hier entweder einem **RoleAList**- oder einem **RoleBList**-Objekt.

Verwendung einer allgemeinen Listenstruktur für alle Rollenlisten: L3

Da die Rollenklassen generiert werden und die Listenstrukturen nur intern verwendet werden, ist eine statische Typprüfung verzichtbar. Daher wird für alle Listenstrukturen die Klasse **RoleList** eingesetzt.

5.3.3 Generierung des Rollenmodells

Zur Integration der Kardinalitäten muß die in Abschnitt 5.2.4 (S. 110) vorgestellte Grammatik zur Definition des Rollenmodells ergänzt werden:

RoleModelDescriptionCA ::= **rolemodel$** *RootRoleRelationship*$^{+}$
RoleRelationship* **$rolemodel**
RootRoleRelationship ::= **root$** *RoleIdentifier*
hasroles$** *RoleDeclaration*$^{+}$ **$root
RoleRelationship ::= **role$** *RoleIdentifier*
hasroles$** *RoleDeclaration*$^{+}$ **$role
RoleDeclaration ::= *RoleIdentifier Cardinality*^{opt}
Cardinality ::= **(** ⟨ *IntValue* | ***** ⟩ **)**
IntValue ::= { **1**,**2**,**3**,... }
RoleIdentifier ::= *UpperCaseLetter*
⟨ *UpperCaseLetter* | *LowerCaseLetter* ⟩*
UpperCaseLetter ::= { **A**,**B**,...,**Z** }
LowerCaseLetter ::= { **a**,**b**,...,**z** }

Prg. 26 zeigt den Inhalt der Beschreibungsdatei für das auf der rechten Seite von Abb. 58 (S. 139) definierte Rollenmodell.

```
1: rolemodel$
2: root$ Person hasroles$ Student $root
3: role$ Student hasroles$ Tutor(4) $role
4: $rolemodel
```

Prg. 26 | Beschreibungsdatei für das rechte Rollenmodell aus Abb. 58 (S. 139)

Die Abbildung Abb. 62 enthält den Aufbau der zu **RoleTutor** korrespondierenden Klasse **Tutor**.

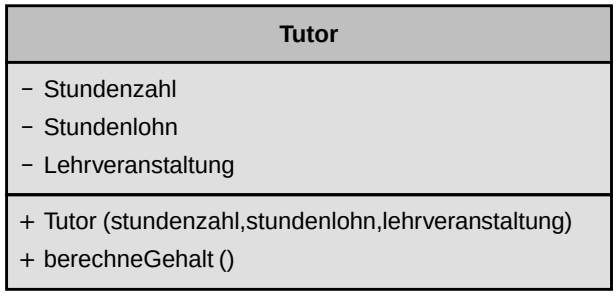


Abb. 62 | Die Klasse **Tutor**

Die Struktur der generierten Rollenklassen ist in Abb. 63, Abb. 64 (S. 145) und Abb. 65 (S. 146) dargestellt.

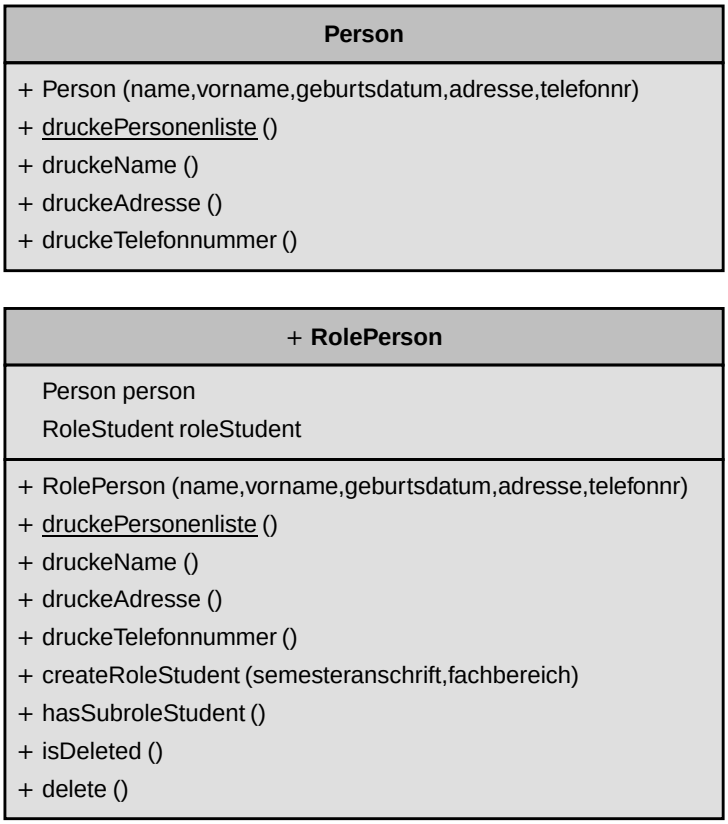


Abb. 63 Struktur der Klasse **RolePerson**

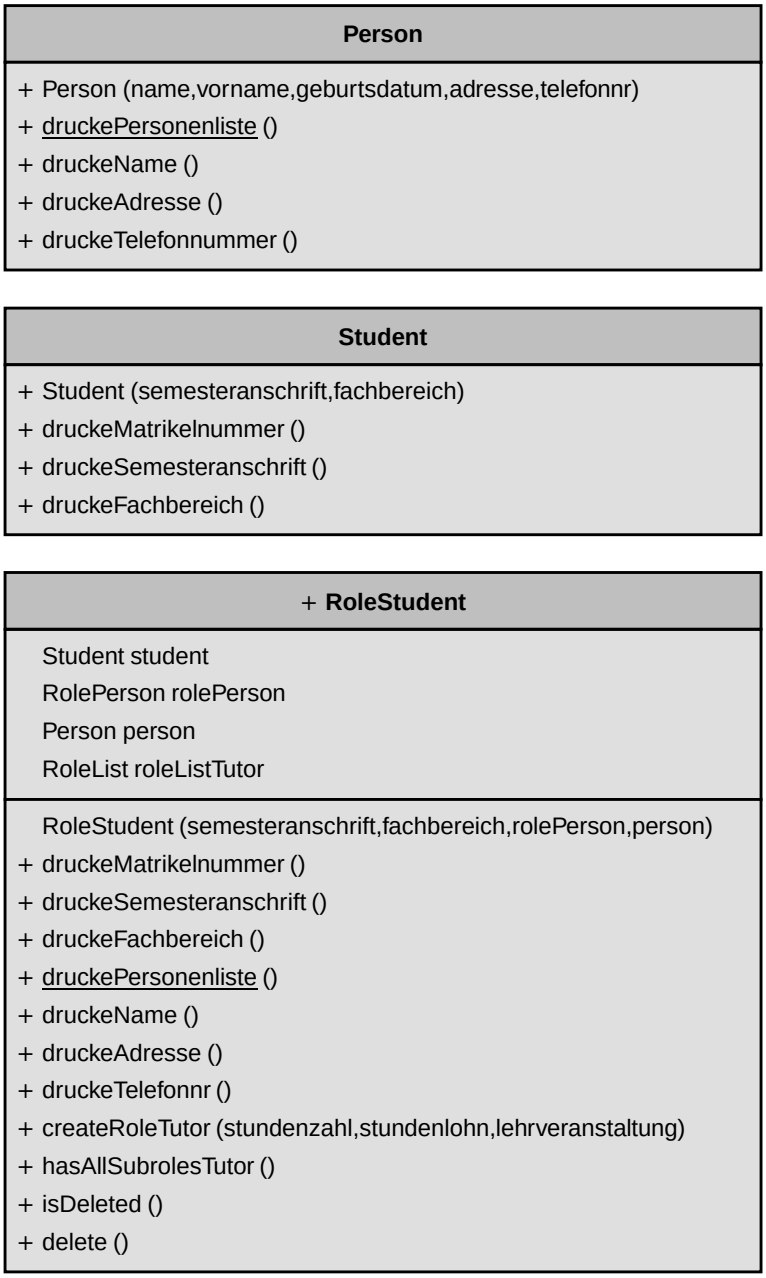


Abb. 64 Struktur der Klasse **RoleStudent**

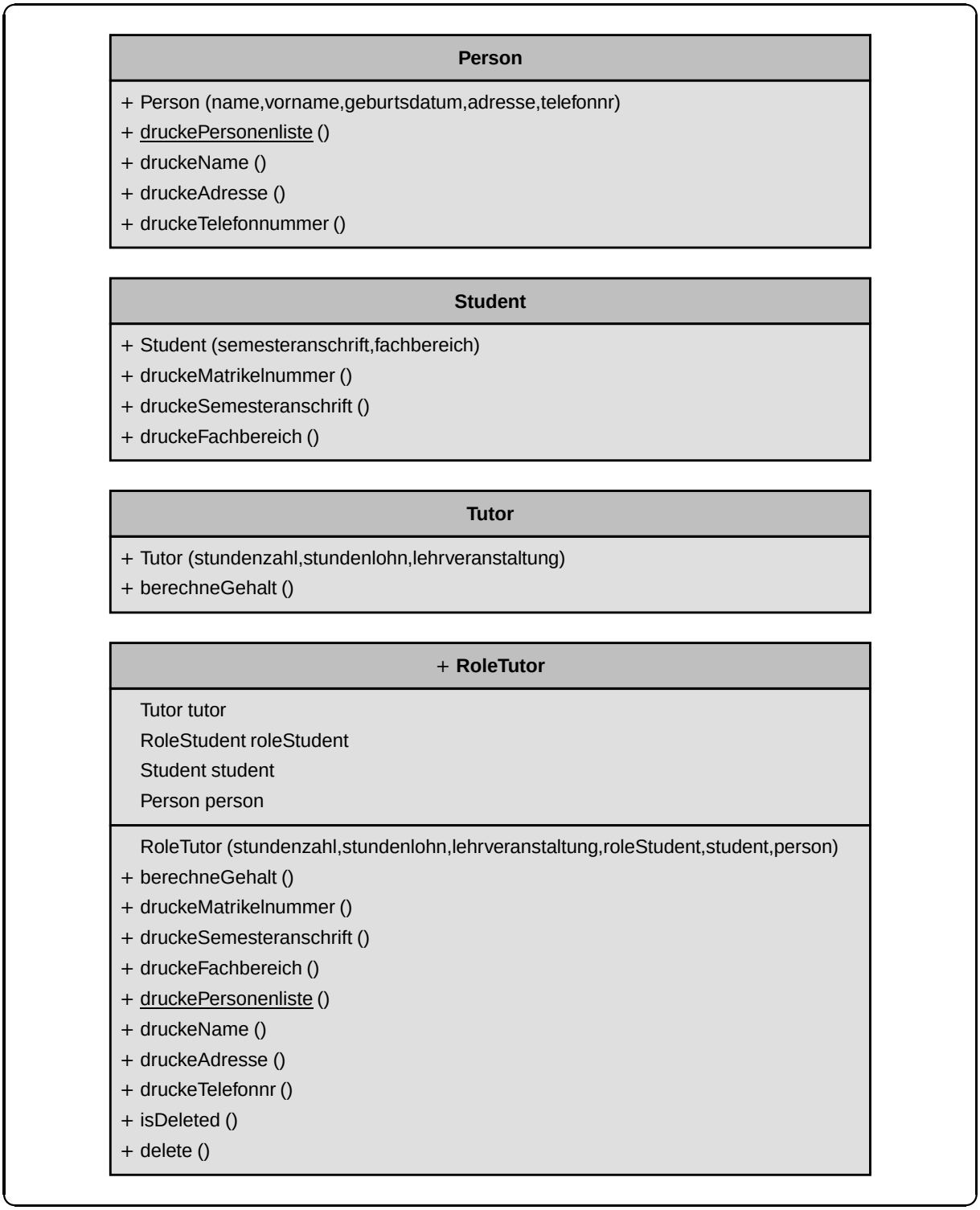


Abb. 65 Struktur der Klasse **RoleTutor**

Der generierte Quelltext der Klassen **RoleStudent** und **RoleTutor** ist in Prg. 27 (S. 147) bzw. Prg. 28 (S. 148) enthalten. Dabei werden neben den Konstruktoren nur die Operationen dargestellt, die von der Einführung der Listenstruktur betroffen sind.

```
 1: package roleModel.cardinality;
 2: import scanner.*;
 3:
 4: public class RoleStudent {
 5:   Student student = null;
 6:   RolePerson rolePerson = null;
 7:   Person person = null;
 8:   RoleList roleListTutor = new RoleList(4);
 9:
10:   RoleStudent ( Adresse semesteranschrift,
11:       String fachbereich, RolePerson rolePerson, Person person ) {
12:     student = new Student ( semesteranschrift,fachbereich );
13:     this.rolePerson = rolePerson;
14:     this.person = person;
15:   }
16:
17:   public RoleTutor createRoleTutor ( double stundenzahl,
18:               double stundenlohn, String lehrveranstaltung )
19:           throws AllSubrolesExist,RoleDeleted {
20:     if ( student==null )
21:        throw new RoleDeleted ( "Student" );
22:     if ( roleListTutor.isFull() )
23:        throw new AllSubrolesExist ( "Tutor" );
24:     RoleTutor roleTutor;
25:     roleTutor = new RoleTutor ( stundenzahl,stundenlohn,
26:        lehrveranstaltung,this,student,person );
27:     roleListTutor.insert(roleTutor);
28:     return roleTutor;
29:   }
30:
31:   public boolean hasAllSubrolesTutor()
32:          throws RoleDeleted {
33:     if ( student==null )
34:        throw new RoleDeleted ( "Student" );
35:     return roleListTutor.isFull();
36:   }
37:
38:   public void delete () throws RoleDeleted {
39:     if ( student==null )
40:        throw new RoleDeleted ( "Student" );
41:     student = null;
42:     person = null;
43:     roleListTutor.deleteAll();
44:     rolePerson.roleStudent = null;
45:     rolePerson = null;
46:   }
47: }
```

Prg. 27 Die Rolle **RoleStudent**

```
 1: package roleModel.cardinality;
 2:
 3: import scanner.*;
 4:
 5: public class RoleTutor implements MultipleRole {
 6:   Tutor tutor = null;
 7:   RoleStudent roleStudent = null;
 8:   Student student = null;
 9:   Person person = null;
10:
11:   RoleTutor ( double stundenzahl, double stundenlohn,
12:       String lehrveranstaltung, RoleStudent roleStudent,
13:       Student student, Person person ) {
14:     tutor = new Tutor ( stundenzahl,stundenlohn,
15:        lehrveranstaltung );
16:     this.roleStudent = roleStudent;
17:     this.student = student;
18:     this.person = person;
19:   }
20:
21:   public void delete () throws RoleDeleted {
22:     if ( tutor==null )
23:        throw new RoleDeleted ( "Tutor" );
24:     tutor = null;
25:     student = null;
26:     person = null;
27:     roleStudent.roleListTutor.remove(this);
28:     roleStudent = null;
29:   }
30: }
```

Prg. 28 Die Rolle **RoleTutor**

5.4 Rollenmodelle mit Mehrfachvererbung

5.4.1 Analysemodell

Grundlagen

Obwohl die Mehrfachvererbung bei modernen objektorientierten Programmiersprachen wie JAVA ([AGH00]) oder C# ([Arc01]) nicht mehr unterstützt wird, stellt sie für die Modellierung von Rollenmodellen ein interessantes Konzept dar, welches nicht von vornherein ausgeschlossen werden sollte.[11]

Analysiert man die Tutorenrolle eines Studenten genauer, stellt sich heraus, daß diese wesentliche Eigenschaften eines Universitätsmitarbeiters besitzt. Daher ist es sinnvoll, die gemeinsamen Eigenschaften der Rollen **RoleTutor**, **RoleWissenschaftlicherMitarbeiter** und **RoleVerwaltungsangestellter** innerhalb der Rolle **RoleUniMitarbeiter** zu konzentrieren. Ein Tutor besitzt dann sowohl die Studenten- als auch die Universitätsmitarbeiterrolle (vgl. Abb. 66).

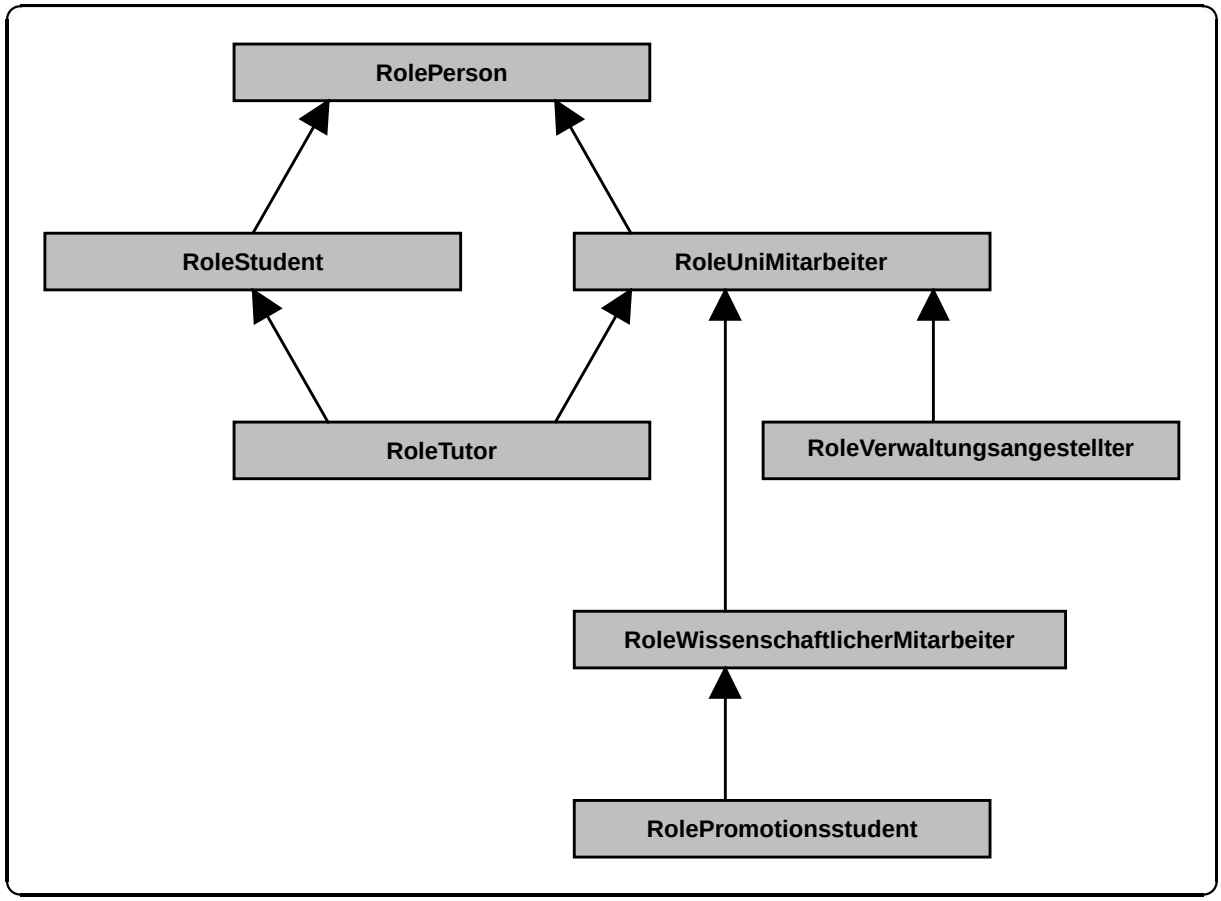


Abb. 66 Universitätsmodell mit Mehrfachvererbung

[11] In [Bal99, S. 54] wird argumentiert, daß die Mehrfachvererbung bei der Erstellung eines Analysemodells im allgemeinen nicht benötigt wird. Diese Aussage könnte unter anderem darauf zurückzuführen sein, daß eine Umsetzung von Rollenmodellen mit Sprachen, die eine statische Typprüfung verwenden, nur sehr schlecht möglich ist ([HK99, S. 62]) und daher Rollenmodelle auch bereits in der Analysephase nicht konsequent eingesetzt werden.

In dem obigen Beispiel wurde angenommen, daß **RoleUniMitarbeiter** eine eigenständige Rolle darstellt, d.h. es gibt Personen, die diese Rolle übernehmen, obwohl sie weder Tutor noch Verwaltungsangestellter noch wissenschaftlicher Mitarbeiter sind. Wenn diese Eigenschaft aus fachlicher Sicht nicht gewünscht ist, dann sollte auf eine eigenständige Rolle **RoleUniMitarbeiter** bei der Modellierung der Rollenhierarchie verzichtet werden. Stattdessen können die gemeinsamen Eigenschaften der verschiedenen Mitarbeiterrollen auf der Ebene der korrespondierenden Klassen über die Einführung einer abstrakten Oberklasse **UniMitarbeiter** berücksichtigt werden. Dies entspricht der in Abb. 57 (S. 137) vorgestellten Struktur. Interessanterweise reduziert sich die Rollenhierarchie aus Abb. 66 (S. 149) damit wieder auf ein Modell mit Einfachvererbung. Die Mehrfachvererbung wurde durch eine Verteilung auf die beiden Vererbungshierarchien, zum einem im Rollenmodell und zum anderen bei den korrespondierenden Klassen, aufgelöst (vgl. Abb. 67).

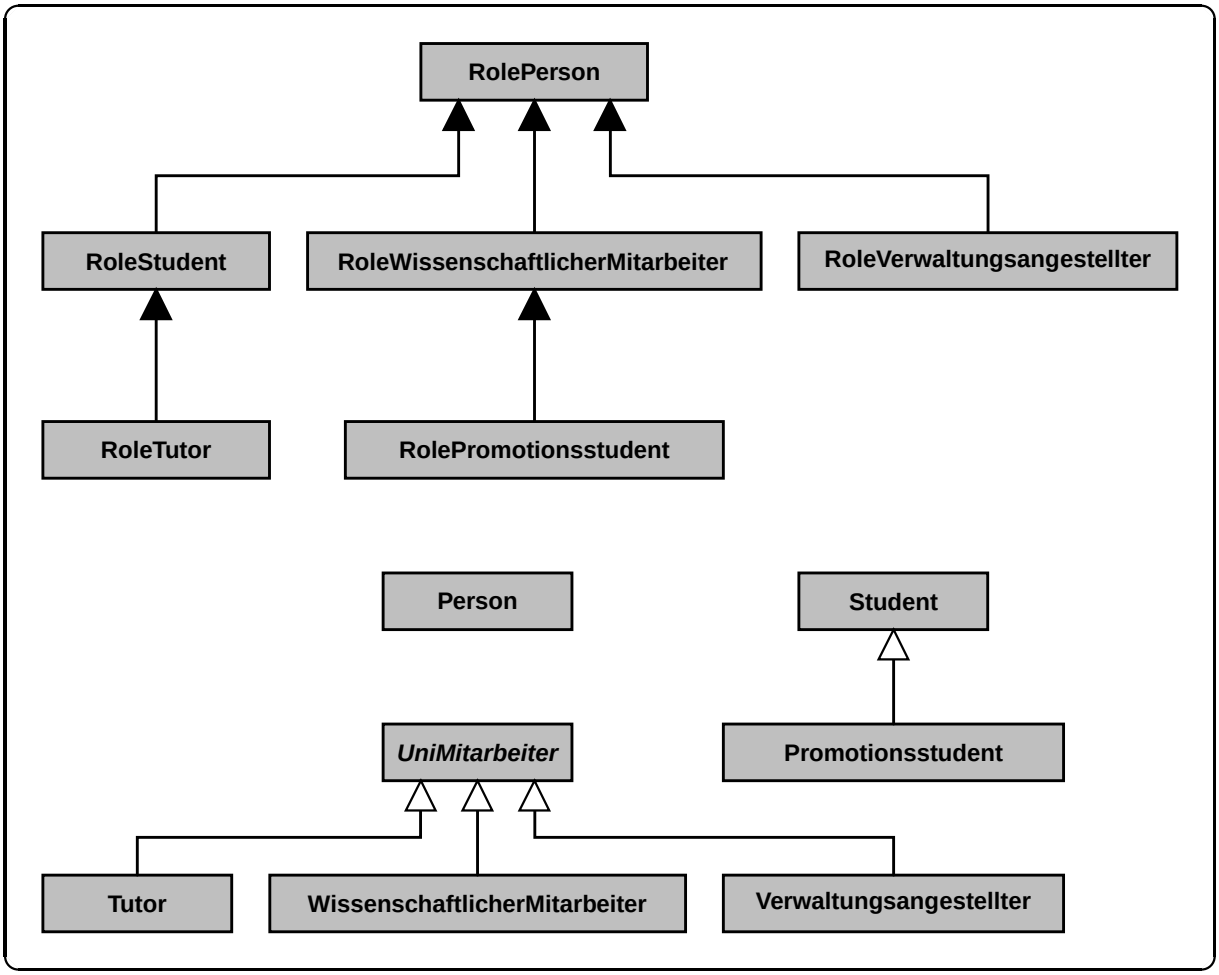


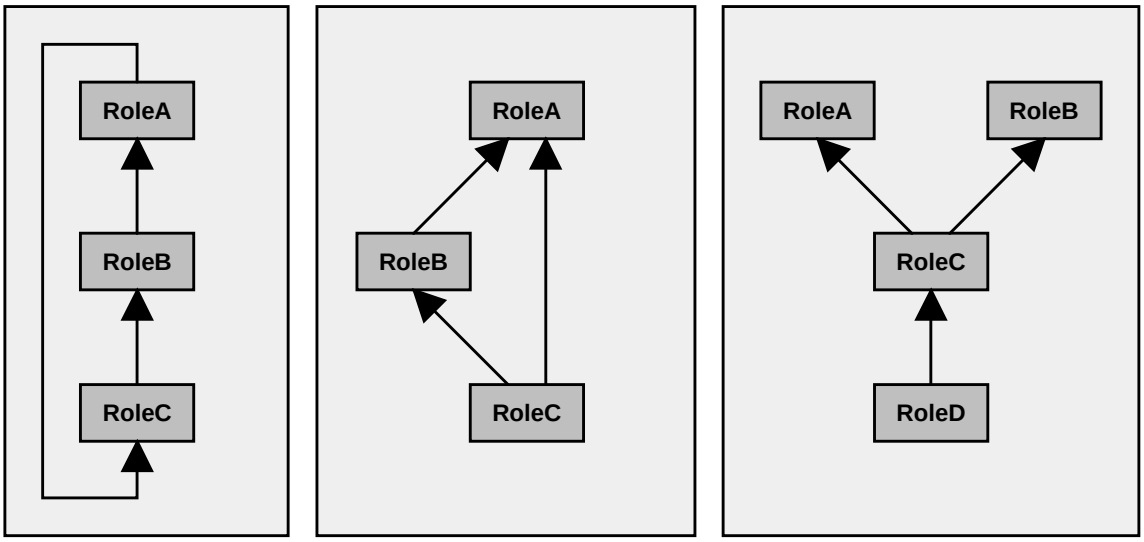
Abb. 67 Auflösung der Mehrfachvererbung beim Universitätsmodell

Für die Definition eines Rollenmodells wird jetzt statt einer Baumstruktur ein ***Graph*** verwendet, der folgende Eigenschaften besitzt:

- Der Graph muß ***zyklenfrei*** sein:
  Besitzt der Graph einen Zyklus, z.B. von **RoleA** über **RoleB** und **RoleC** nach **RoleA**, dann

gilt: **RoleA** *ist eine Rolle von* **RoleC** und **RoleC** *ist eine Rolle von* **RoleA** (vgl. Abb. 68 (a)). Dies widerspricht der Generalisierungs- bzw. Spezialisierungseigenschaft, die eine Rollenbeziehung auf semantischer Ebene erfüllen soll.

- Der Graph hat keine ***transitiven*** Rollenbeziehungen:
  Dies ist gleichbedeutend mit der Forderung, daß eine Rolle nicht gleichzeitig direkte und indirekte Unterrolle einer anderen Rolle sein darf. In Abb. 68 (b) besitzt **RoleC** diese Eigenschaft, die eine unnötige Mehrfachvererbung darstellt und das Rollenmodell nur komplexer als notwendig gestaltet.
- Der Graph darf ***mehrere Wurzelrollen*** besitzen:
  Eine Wurzelrolle hat die Eigenschaft, keine Oberrollen zu besitzen. Eine Beschränkung der Rollenmodelle auf eine Wurzelrolle hätte den einzigen Vorteil, daß von einem Wurzelrollenobjekt aus alle existierenden Unterrollen (rekursiv) gelöscht werden können, da in diesem Fall grundsätzlich immer alle Unterrollen von der Wurzelrolle aus erreichbar sind. Läßt man mehrere Wurzelrollen zu, gilt diese Eigenschaft nicht mehr. In Abb. 68 (c) können z.B. ausgehend von **RoleA** neben dem **RoleA**-Objekt nur die (eventuell vorhandenen) **RoleC**- und **RoleD**-Objekte gelöscht werden. Das **RoleB**-Objekt, welches existieren muß, wenn ein **RoleC**-Objekt vorhanden war[12], bleibt bestehen und kann den Ausgangspunkt einer neuen Rollenhierarchie bilden.



(a) Zyklus (b) transitive Rollenbeziehung (c) mehrere Wurzelrollen

Abb. 68 Eigenschaften von Rollenmodellen mit Mehrfachvererbung

[12] Es ist zu beachten, daß ein Unterrollenobjekt nur erzeugt werden kann, wenn alle direkten und indirekten Oberrollenobjekte existieren.

Eine weitere Anforderung könnte sein, bei der Definition eines Rollenmodells nur zusammenhängende[13] Graphen zuzulassen. Da ein nicht zusammenhängender Graph aber lediglich zur Folge hat, daß über ihn mehrere, vollständig unabhängige Rollenmodelle spezifiziert werden[14], wird auf diese Anforderung verzichtet.

Die Erzeugung von Unterrollenobjekten

Für die Erzeugung von Unterrollenobjekten stellt sich die Frage, ob sie ausgehend von allen direkten Oberrollenobjekten erlaubt sein soll. Betrachtet man die Rolle **RoleTutor** des Rollenmodells aus Abb. 66 (S. 149), so wird bei der Erstellung des Fachkonzepts, welches den betreffenden Ausschnitt der realen Welt in einem objektorientierten Modell abbilden soll, wahrscheinlich gefordert werden, daß nur Studenten und nicht Universitätsmitarbeiter eine Tutorenrolle übernehmen können. Damit steht man wie bei dem Thema *Vererbung zwischen korrespondieren Klassen* (vgl. S. 136) vor der Entscheidung, ob man derartige Restriktionen auf der Ebene des Rollenmodells formuliert oder von der Anwendungsebene kontrollieren läßt. Auf diesen Aspekt wird näher im Abschnitt 5.6 (S. 174) eingegangen. Hier soll zunächst die Eigenschaft gelten, daß `createRoleX`-Operationen in allen direkten Oberrollen von **RoleX** vorhanden sind.

Bei Rollenmodellen mit Einfachvererbung gibt es drei Eigenschaften, die erfüllt sein müssen, damit eine Unterrolle erzeugt werden darf:

- E1: Für alle im Rollenmodell definierten Oberrollen müssen entsprechende Rollenobjekte existieren.
- E2: Die durch die Kardinalitätsspezifikation vorgegebene Obergrenze der erlaubten Unterrollenobjekte ist noch nicht erreicht.
- E3: Durch die Erzeugung eines Unterrollenobjekts darf die Konsistenz der Rollenobjektstruktur nicht verletzt werden. Als Konsistenzanforderung gilt hierbei, daß einem Unterrollenobjekt genau ein Objekt jeder Oberrollenklasse zugeordnet ist.

Die erste Anforderung wird bei der Einfachvererbung durch die Baumstruktur sowie die Struktur der `createRole`- und `delete`-Operationen garantiert. Liegt eine Mehrfachvererbung vor, dann ist diese Anforderung erfüllt, sofern die `createRoleY`-Operation für eine Unterrolle **RoleY** in der Klasse `RoleX` als Parameter Referenzen auf alle anderen direkten Oberrollen von **RoleY** besitzt und alle beim Aufruf übergebenen Referenzen auf existierende Rollenobjekte verweisen. In dem Rollenmodell aus Abb. 69 (S. 153) wäre dies für die `createRoleF`-Operation aus der Rollenklasse `RoleD` eine `RoleE`-Referenz. Da aus der Existenz der direkten Oberrollenobjekte folgt, daß alle ihre direkten (und indirekten) Oberrollenobjekte ebenfalls existieren, ist damit E1 für die Mehrfachvererbung sichergestellt.

Die zweite Anforderung ist der bei Mehrfachvererbung ähnlich einfach wie bei der Einfachvererbung zu erfüllen: es muß bei allen direkten Oberrollenobjekten überprüft werden, daß die Anzahl der erlaubten Unterrollenobjekte noch nicht erreicht ist.

Die dritte Anforderung ist bei der Einfachvererbung ebenfalls automatisch durch die vorgegebene Baumstruktur erfüllt, sofern die zweite Anforderung eingehalten wird. Wie das folgende Beispiel zeigt, gilt bei Verwendung der Mehrfachvererbung diese Aussage nicht mehr, weil jetzt

[13] Ein gerichteter Graph ist zusammenhängend, falls es zwischen je zwei Knoten einen (ungerichteten) Weg gibt, wenn man die Richtung der Kanten wegläßt ([Eng93, S. 288]).

[14] Pro zusammenhängendem Teilgraphen entsteht ein eigenes Rollenmodell.

ein azyklischer Graph bei der Rollenmodellstruktur vorliegt. Für das Rollenmodell aus Abb. 69 sollen die in Abb. 70 dargestellten Rollenobjekte vorliegen. Eine Kante von x nach y hat dabei die Bedeutung *ist direktes Oberrollenobjekt von*. Wird nun die `createRoleF`-Operation

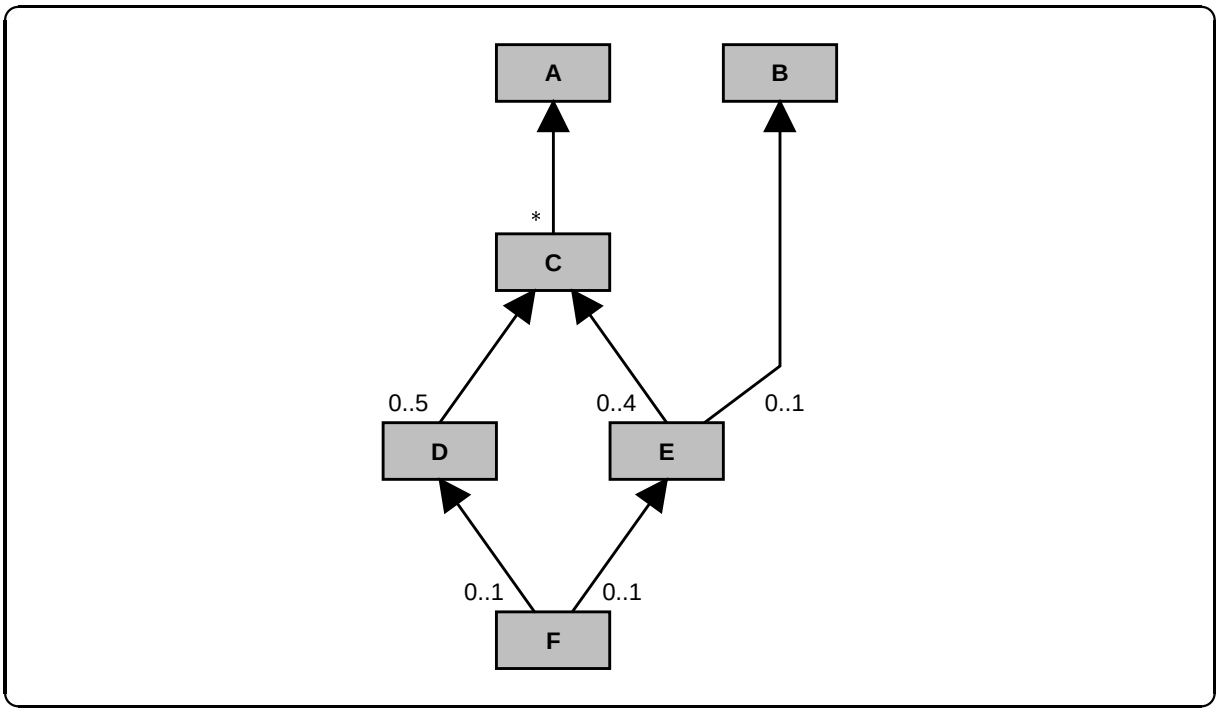


Abb. 69 Rollenmodell für das Beispiel aus Abb. 70

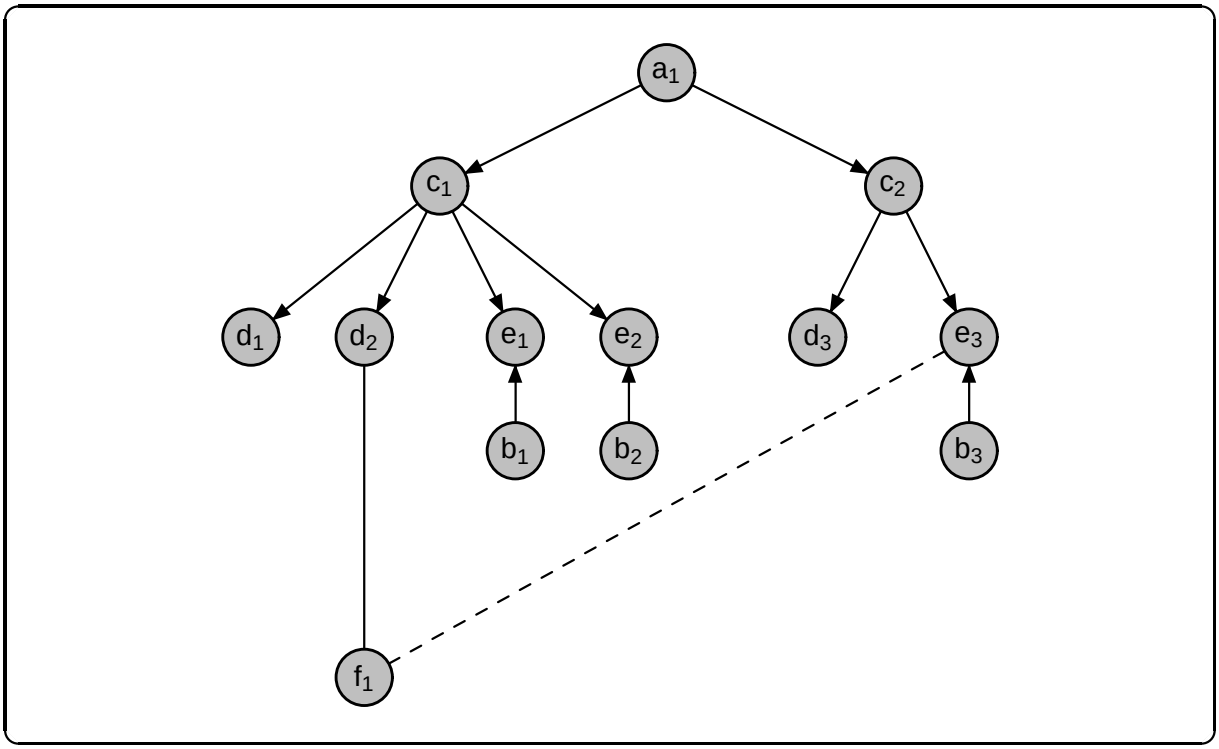


Abb. 70 Die Erzeugung eines Rollenobjekts für das Rollenmodell aus Abb. 69

des Rollenobjekts d_2 aufgerufen, ist ein Oberrollenobjekt der Rollenklasse **RoleE** auszuwählen. Von den drei existierenden Objekten e_1, e_2 und e_3 kommen aber nur e_1 und e_2 in Frage. Würde, wie in Abb. 70 (S. 153) durch die gestrichelte Linie dargestellt, e_3 verwendet, dann entstände eine inkonsistente Rollenobjektstruktur: aufgrund des Aufrufs der `createRoleF`-Operation auf d_2 erhielte f_1 die (indirekte) Oberrolle c_1 des Typs **RoleC**. Durch eine Beziehung zu e_3 käme c_2 hinzu, was die Anforderung verletzt, daß ein Rollenobjekt immer nur genau ein Oberrollenobjekt einer Rollenklasse besitzen darf. Auf die Struktur eines geeigneten Tests zur Sicherstellung dieser Anforderung wird in Abschnitt 5.4.2 (S. 160) eingegangen.

Die Redefinition von Operationen

Wie bei der Einfachvererbung soll eine Rollenklasse wieder alle öffentlichen Operationen aus ihrer korrespondierenden Klasse sowie aus allen korrespondierenden Klassen der direkten und indirekten Oberrollen besitzen.

Mit der Einführung der Mehrfachvererbung in eine Programmiersprache hat man das Problem der Namenskonflikte. Dieses betrifft sowohl Attribut- als auch Operationsnamen. In der Klasse **F** aus Abb. 71 tritt ein Namenskonflikt für das Attribut `a` auf: `a` ist in den direkten Oberklassen **C** und **E** definiert sowie in den indirekten Oberklassen **A** und **B**. Zusätzlich besitzt `a` (z.B.) in **C** und **E** noch unterschiedliche Typen.

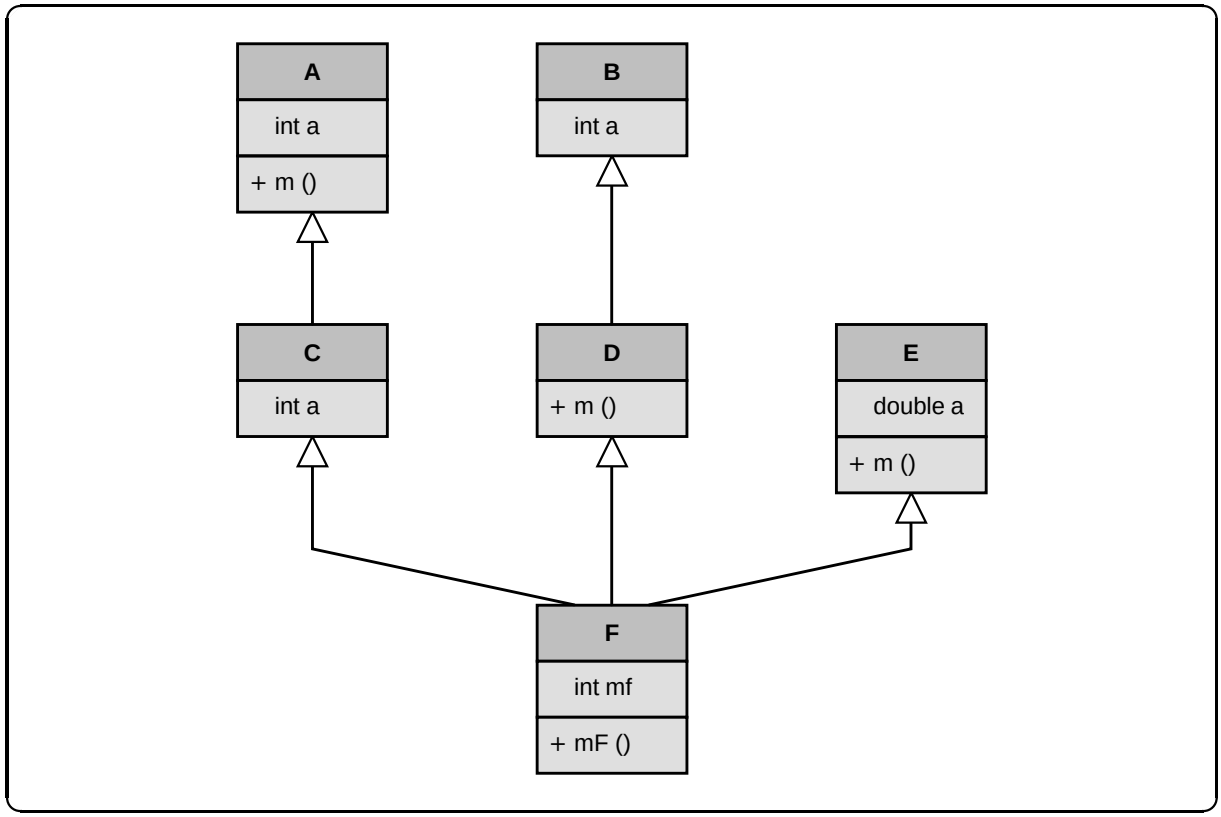


Abb. 71 Namenskonflikte bei der Mehrfachvererbung

Betrachtet man die Definition einer Rollenhierarchie, so läßt sich feststellen, daß für Attribute das Problem der Namenskonflikte nicht vorhanden ist. Die Rollenklassen übernehmen keine At-

tribute aus den korrespondierenden Klassen; alle in den Rollenklassen vorhandenen Attribute dienen allein dem Aufbau der notwendigen Beziehungen zwischen den Rollenobjekten. An dieser Stelle ist zu beachten, daß zwar zwischen den Rollenklassen selbst auf der Entwurfs- und Implementierungsebene keine Vererbungsbeziehungen bestehen, es jedoch bei einer Übernahme von Attributen der korrespondierenden Klassen zu Namenskonflikten hätte kommen können, da die korrespondierenden Klassen zur Definition des Rollenmodells in einem azyklischen Graphen angeordnet werden dürfen. Bei den korrespondierenden Klassen kann es ebenfalls keine Namenskonflikte geben, da sie (wenn überhaupt) unter Verwendung der Einfachvererbung realisiert werden.

Für die Operationen gilt die obige Aussage allerdings nicht mehr. Da eine Rollenklasse alle öffentlichen Operationen der direkten und indirekten Oberrollenklassen übernimmt, sind Namenskonflikte möglich. Nimmt man die in Abb. 71 (S. 154) dargestellen Klassen und Vererbungsstrukturen als Grundlage für ein Rollenmodell, so ensteht für die Operation `m` bezogen auf die Klasse **RoleF** ein Namenskonflikt: `m` ist in den direkten Oberrollen **RoleE** und **RoleD** sowie in der indirekten Oberrolle **RoleA** definiert. Damit ist nicht eindeutig festgelegt, auf welche dieser drei Operationen in **RoleF** Bezug genommen werden soll.

Im Programmiersprachenbereich existieren für das Problem der Namenskonflikte zwei grundlegende Lösungsalternativen ([Boo94]):

- Namenskonflikte werden verboten.
  Um diese Regel einzuhalten, sind zwei Vorgehensweisen denkbar:
  - Erkennt der Compiler einen Namenskonflikt, dann muß in den Oberklassen durch Umbenennung wieder Eindeutigkeit hergestellt werden. Dieser Ansatz widerspricht jedoch einer baukastenartigen Kombination von Oberklassen ([Mey90, S. 266]). Außerdem müßte beim konkreten Auftreten eines Konflikts der Quelltext der Oberklassen vorliegen, um durch Namensänderungen die Eindeutigkeit herzustellen.
  - Bei der Definition einer Unterklasse wird der Namenskonflikt durch eine entsprechende Umbenennung aufgelöst. Dieser Ansatz wird in EIFFEL durch die Einführung der **`rename`**-Klausel ([Mey90, S. 266]) verwendet.
- Angabe der konkret zu verwendenden Oberklasse beim Auftreten eines Namenskonflikts.
  Bei jedem Zugriff auf ein Oberklassenattribut oder eine Oberklassenoperation, der zu einem Namenskonflikt führt, muß der Anwender die Oberklasse explizit angeben, aus der das Attribut bzw. die Operation ausgewählt werden soll. Dieser Ansatz wird (z.B.) in C^{++} benutzt.

Für die Definition von Rollenmodellen wird hier der erste Ansatz verwendet. Falls Namenskonflikte auftreten, wird das Rollenmodell als fehlerhaft betrachtet. Die Definition des Rollenmodells ist ein Bestandteil der Analysephase des Software-Entwicklungsprozesses und dient zur Festlegung des Fachkonzepts. Daher sollte der Systemanalytiker darauf achten, daß hier keine Mehrdeutigkeiten auftreten, da das Analysemodell die Fachkonzeptfunktionalität des Software-Systems präzise beschreiben soll.

Es bleibt an dieser Stelle noch die Frage zu klären, wann genau ein Namenskonflikt bei den Operationen des Rollenmodells vorliegt. Wäre beispielsweise in der Klasse **F** aus Abb. 71 (S. 154) die Operation `m` ebenfalls definiert, läge kein Namenskonflikt vor, da jetzt alle `m`-Operationen durch diese Redefinition verdeckt werden. Für eine präzise Definition eines Namenskonflikts läßt sich der folgende Ansatz verwenden:

- Das Rollenmodell wird als gerichteter, azyklischer Graph aufgefaßt.
  - Die Knoten des Graphen stellen die Rollen des Rollenmodells dar.
  - Eine Kante vom Knoten R_1 zum Knoten R_2 bedeutet, daß R_2 eine direkte Unterrolle von R_1 ist.
- Für eine Rolle R sei die Menge $TG(R)$ wie folgt definiert:
  - $R \in TG(R)$.
  - Es gibt einen Pfad von N nach $R \;\Rightarrow\; N \in TG(R)$.

  $TG(R)$ enthält somit neben R alle direkten und indirekten Oberrollen.
- Für eine Rolle R und eine Operation m ist $TG(R,m)$ die Menge aller Knoten (d.h. Rollen) aus $TG(R)$, die die Operation m definieren oder redefinieren.[15]
- Das Rollenmodell enthält genau dann einen Namenskonflikt bezüglich der Operation m, wenn gilt:
  - Es gibt mindestens eine Rolle R mit
    - $TG(R,m) \neq \emptyset$ und
    - die topologische Sortierung der Knoten aus $TG(R,m)$ besitzt ***kein eindeutiges Maximum*** $TG(R,m)_{\max}$.

In einem gerichteten Graphen gilt für zwei Knoten v und w die Relation $v < w$ genau dann, wenn von v nach w ein Pfad existiert. Da die betrachteten Graphen azyklisch sind, kann nicht gleichzeitig $v < w$ und $w < v$ gelten. Allerdings ist es möglich, daß weder $v < w$ noch $w < v$ gilt, da über den Graphen lediglich eine ***Halbordnung*** ([BS81, S. 600]) definiert wird. Liegt ein Graph mit den Knoten $V = \{ v_1, v_2, \ldots v_n \}$ vor, so ist ein Knoten $v_{\max} \in V$ genau dann ein eindeutiges Maximum, wenn gilt: $v_{\max} > v_i$ mit $i \in \{ 1, \ldots n \}$ und $i \neq \max$.

Der linke Graph in Abb. 72 (S. 157) definiert ein Rollenmodell, bei dem für die Operation m ein Namenskonflikt auftritt, da (z.B.) für den Knoten G gilt:

$$
\begin{aligned}
&TG(G) = \{A, B, C, D, E, F, G\} \\
&TG(G,m) = \{A, B, C, F\} \neq \emptyset \\
&A < C \quad B < F \quad A < F
\end{aligned}
$$

Da weder $C < F$ noch $F < C$ gilt, liegt kein eindeutiges Maximum vor.

Dagegen besteht für die Operation m in dem rechten Graphen aus Abb. 72 (S. 157) kein Namenskonflikt:

$$
\begin{array}{lll}
TG(A,m) &= \{A\} & TG(A,m)_{\max} = A \\
TG(B,m) &= \emptyset & \\
TG(C,m) &= \{A\} & TG(C,m)_{\max} = A \\
TG(D,m) &= \{A\} & TG(D,m)_{\max} = A \\
TG(E,m) &= \{A\} & TG(E,m)_{\max} = A \\
TG(F,m) &= \{A,F\} & TG(F,m)_{\max} = F \\
TG(G,m) &= \{A,F,G\} & TG(G,m)_{\max} = G \\
TG(H,m) &= \{A,F,G\} & TG(H,m)_{\max} = G
\end{array}
$$

[15] Es reicht also nicht aus, wenn R die Operation m von einer ihrer Oberrollen erbt.

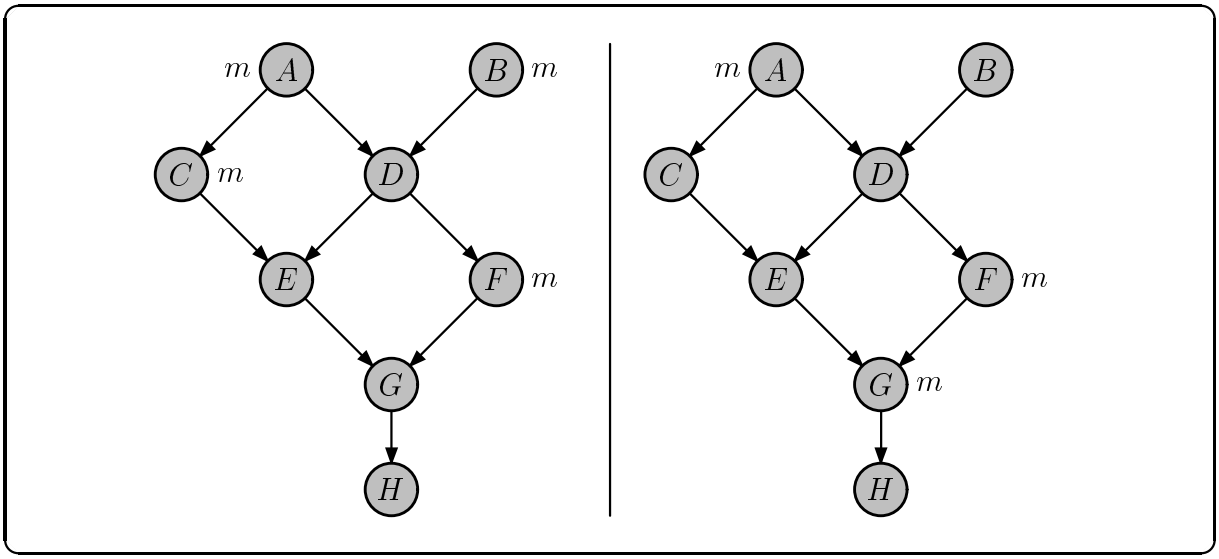


Abb. 72 Ermittlung von Namenskonflikten

Die Semantik der restlichen Verwaltungsoperationen

Im Gegensatz zur `createRole`-Operation ändert sich die Semantik der Operationen `delete`, `isDeleted` und `hasSubrole` im Vergleich zur Einfachvererbung nicht.

Konsistenz der Kardinalitäten

Bei Rollenmodellen mit Einfachvererbung gibt es keine Konsistenzprobleme bei der Wahl der Unterrollenkardinalitäten, da eine Unterrolle immer genau eine direkte Oberrolle besitzt. Bei Verwendung der Mehrfachvererbung besteht dagegen die Möglichkeit, inkonsistente Kardinalitäten zu spezifizieren. Bei dem Rollenmodell aus Abb. 73 ist es nicht möglich, für b_1 eine zweite Unterrolle des Typs **RoleD** zu erzeugen, da c_1 nur ein **RoleD**-Unterrollenobjekt besitzen darf und aufgrund der gemeinsamen Oberrolle **RoleA** sowie der 0..1 Kardinalität zwischen **A** und **C** auch kein anderes Objekt des Typs **RoleC** verwendbar ist (vgl. E2 S. 152).

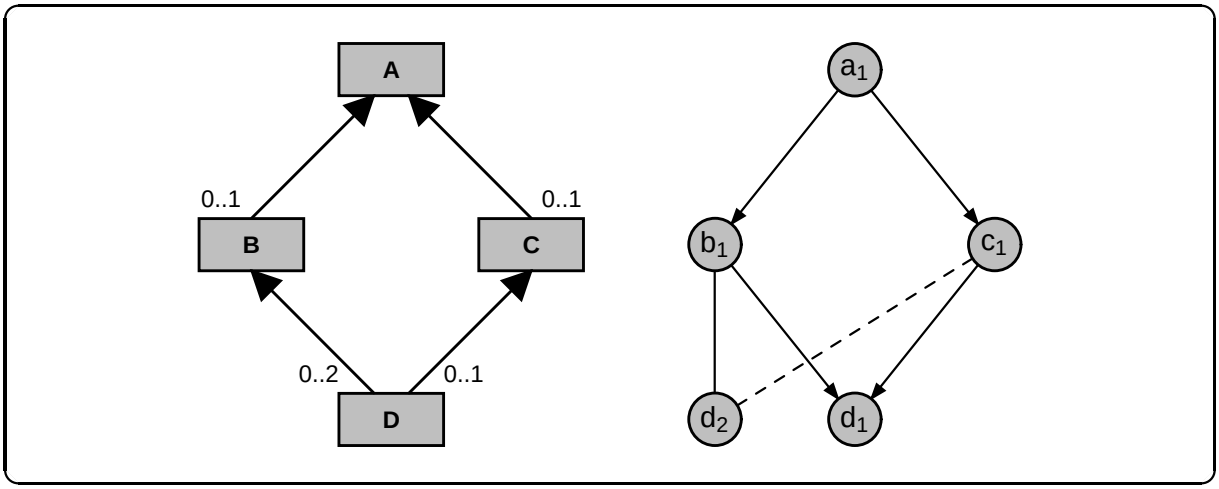


Abb. 73 Inkonsistente Kardinalitäten bei der Mehrfachvererbung

Interessanterweise tritt diese Inkonsistenz nicht mehr auf, wenn die Beziehung zwischen **A** und **C** weggelassen wird. Jetzt kann d_2 als direktes Oberrollenobjekt c_2 verwenden (vgl. Abb. 74).

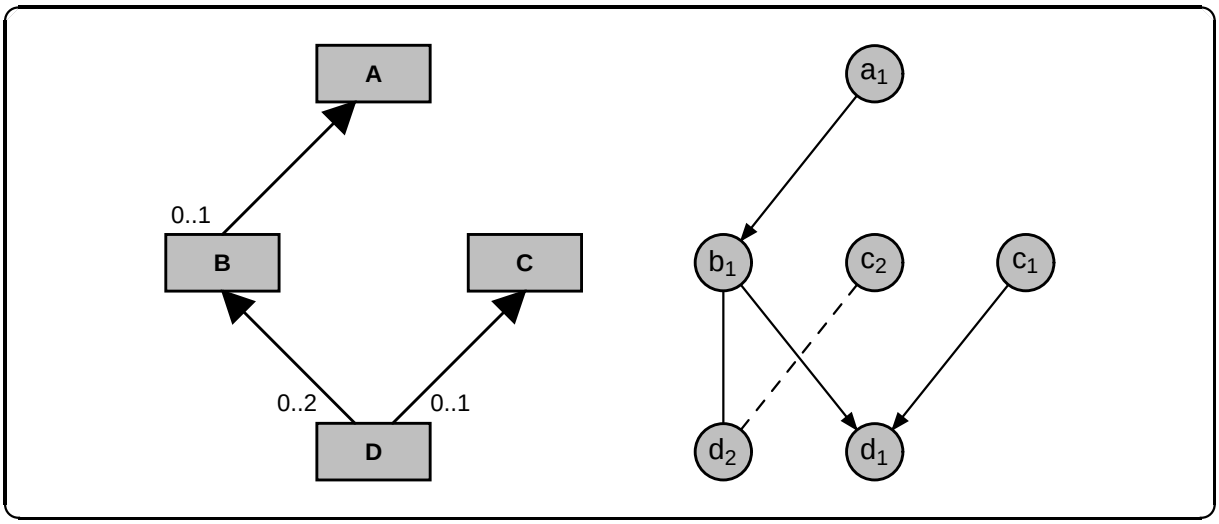


Abb. 74 Die konsistente Variante des Rollenmodells aus Abb. 73 (S. 157)

Durch das Verbot transitiver Rollenbeziehungen (vgl. Abb. 68 (S. 151)) werden bestimmte Inkonsistenzen von vornherein vermieden. Für das Rollenmodell aus Abb. 75 erlaubt die Kardi-

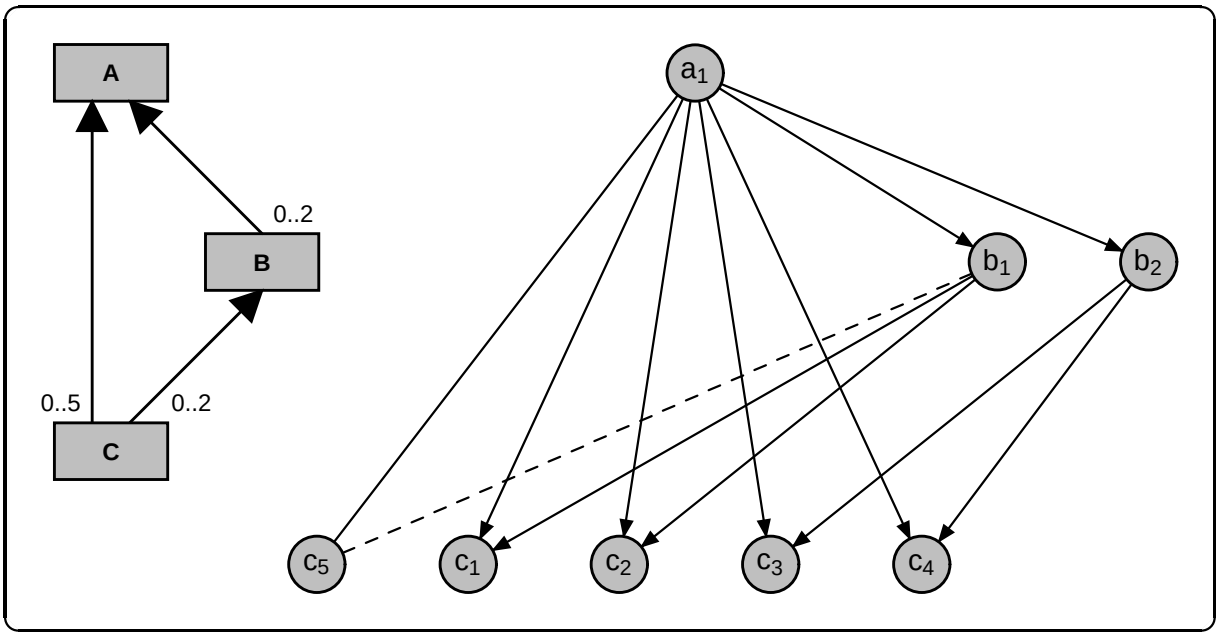


Abb. 75 Inkonsistente Kardinalitäten bei transitiven Rollenbeziehungen

nalität 0..5 zwischen **A** und **C** noch die Erzeugung des Unterrollenobjekts c_5. Da aber keine Beziehung mehr zu einem **RoleB**-Objekt aufgebaut werden kann, muß die Unterrollenobjekterzeugung abgebrochen werden. Durch den Verzicht auf die transitive Rollenbeziehung wird automatisch auch diese Inkonsistenz beseitigt.

Eine Konsistenzprüfung der Kardinalitäten des Rollenmodells vor der Generierung der Rollenklassen erfolgt nicht. Stattdessen wird zur Laufzeit durch die Kontrolle der Eigenschaften E1 bis E3 (vgl. S. 152) sichergestellt, daß keine inkonsistenten Rollenobjekte auftreten, selbst wenn bezüglich der Kardinalitäten ein inkonsistentes Rollenmodell vorliegt.

Weitere Eigenschaften von Rollenmodellen mit Mehrfachvererbung

Analog zu den Rollenmodellen mit Einfachvererbung werden die folgenden Modellierungskonzepte unterstützt:

- Assoziationen zwischen den korrespondierenden Klassen.
- Die Anwendung des Vererbungskonzepts zur Definition der korrespondierenden Klassen.

5.4.2 Entwurfsmodell

Beziehungen zwischen den Rollenklassen und korrespondierenden Klassen

Durch die Einführung der Mehrfachvererbung kann ein Rollenobjekt mehrere direkte Oberrollen besitzen. Ein Unterrollenobjekt muß daher Beziehungen zu allen direkten Oberrollenobjekten verwalten. In Abb. 76 ist die Umsetzung des Rollenmodells aus Abb. 69 (S. 153) dargestellt. Ein **RoleE**-Objekt besitzt hier Referenzen auf ein **RoleC**- und ein **RoleB**-Objekt. Wie bei einem Rollenmodell mit Einfachvererbung besitzt ein Rollenobjekt Beziehungen zu korrespondierenden Objekten aller (direkten und indirekten) Oberrollen sowie zu direkten Unterrollenobjekten.

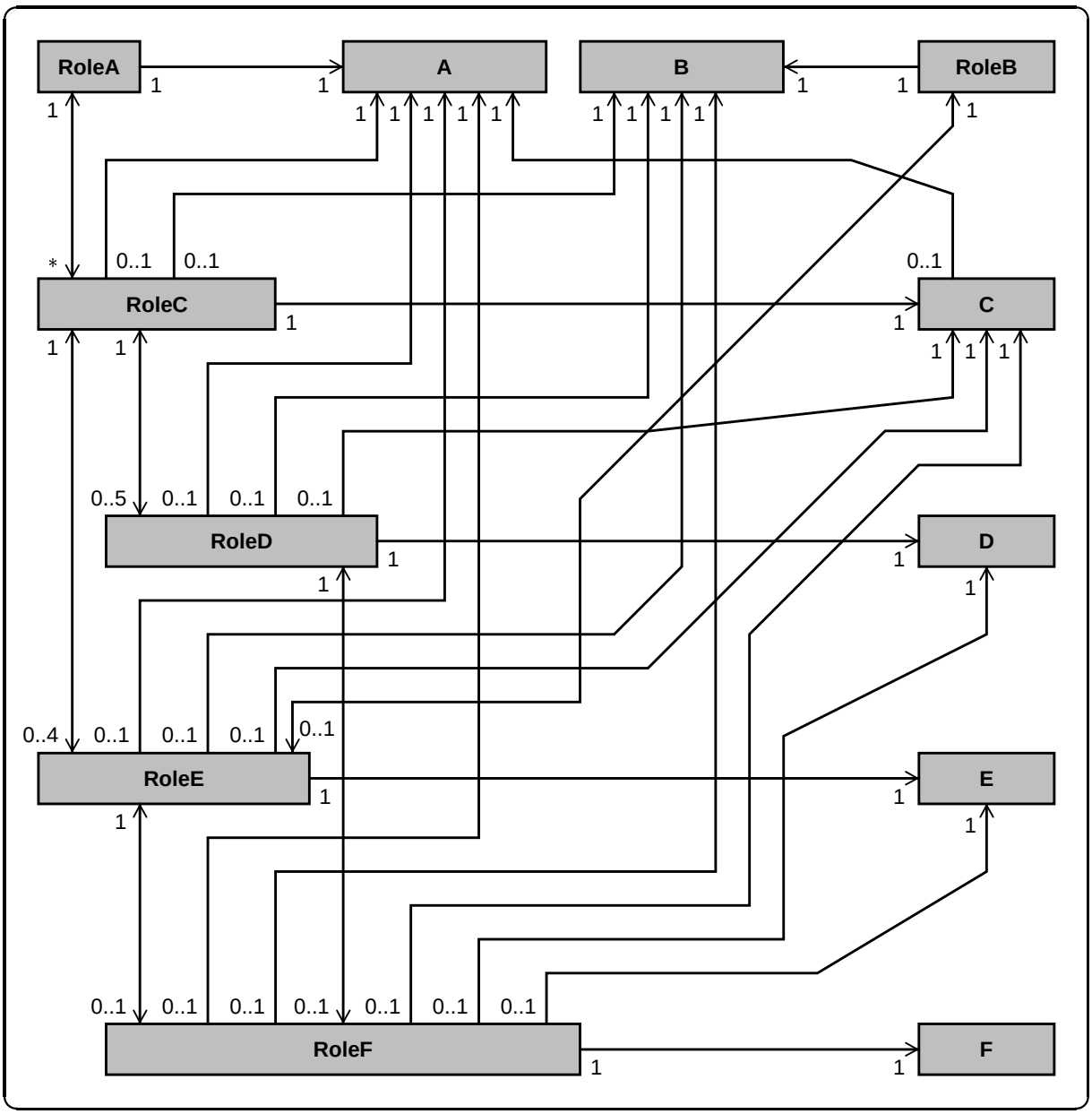


Abb. 76 Klassendiagramm des über Abb. 69 (S. 153) definierten Rollenmodells

Änderungen bei den Konstruktoren

Die Konstruktoren für Wurzelrollen haben dieselbe Struktur wie diejenigen für Rollenmodelle mit Einfachvererbung.

Bei den Konstruktoren für Unterrollen ist zu berücksichtigen, daß sie jetzt die Beziehungen zu (eventuell) mehreren Oberrollenobjekten aufbauen müssen. Daher reicht es nicht mehr aus, nur eine Referenz auf die aufrufende Oberrolle sowie Referenzen auf die korrespondierenden Objekte zu übergeben (vgl. S. 102). Stattdessen besitzt ein Konstruktor hier Referenzen auf alle direkten Oberrollenobjekte als Parameter. Über diese Referenzen lassen sich dann auch die Beziehungen zu den korrespondierenden Objekten aller Oberrollen aufbauen. Prg. 29 zeigt für die Klasse **RoleE** aus dem in Abb. 69 (S. 153) definierten Rollenmodell die Attribute zur Realisierung der erforderlichen Beziehungen sowie den Konstruktor.

```
 1:    E e = null;
 2:
 3:    RoleB roleB = null;
 4:    RoleC roleC = null;
 5:
 6:    B b = null;
 7:    C c = null;
 8:    A a = null;
 9:
10:    RoleF roleF = null;
11:
12:    RoleE ( RoleB roleB, RoleC roleC ) {
13:      this.roleB = roleB;
14:      this.roleC = roleC;
15:      this.b = roleB.b;
16:      this.c = roleC.c;
17:      this.a = roleC.a;
18:      this.e = new E();
19:    }
```

Prg. 29 Der Konstruktor der Klasse **RoleE** für das Rollenmodell aus Abb. 69 (S. 153)

Struktur eines Tests zur Erfüllung der Eigenschaft E3

Eine der aufwendigsten Änderungen beim Übergang von Rollenmodellen mit Einfachvererbung zu Rollenmodellen mit Mehrfachvererbung betrifft die Erfüllung der Anforderung E3 (vgl. S. 152). Durch die Erzeugung eines neuen Unterrollenobjekts darf die Konsistenz der Rollenobjektstruktur nicht verloren gehen. Daher muß bei der Erzeugung eines Unterrollenobjekts sichergestellt werden, daß nicht Beziehungen zu unterschiedlichen Oberrollenobjekten desselben Typs entstehen (vgl. Abb. 70 (S. 153)). Um diese Anforderung zu überprüfen reicht es aus, die Mengen der korrespondierenden Objekte der direkten Oberrollenobjekte des neuen Unterrollenobjekts x zu betrachten. Da das Rollenobjekt vor der Erzeugung von x in einem konsistenten Zustand ist, müssen alle betroffenen direkten Oberrollenobjekte y die Eigenschaft E3 erfüllen.

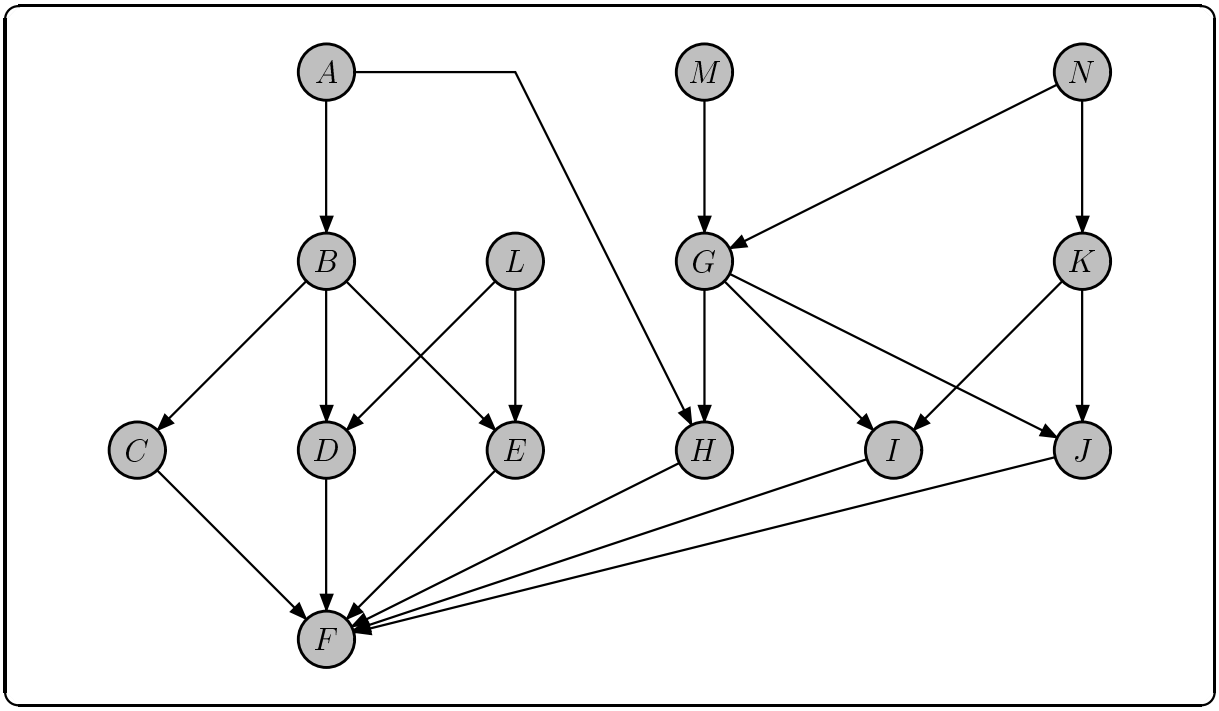


Abb. 77 Das Beispielrollenmodell zur Ermittlung eines Tests für E3

Für das Rollenmodell aus Abb. 77 muß daher beispielsweise gelten: $d.a = b.a$, wenn mit $x.y$ die Referenz auf das korrespondierende Objekt y des Typs Y im Rollenobjekt x des Typs X bezeichnet wird. Eine hinreichende Menge von Vergleichsoperationen entsteht somit, wenn man die Referenzenmengen der direkten Oberrollenobjekte des neu zu erzeugenden Unterrollenobjekts miteinander vergleicht. Sei $SR(X)$ die Menge aller Oberrollen der Rolle X. Wird für das Rollenmodell aus Abb. 77 ein neues Unterrollenobjekt des Typs E erzeugt, dann sind die folgenden SR-Mengen zu berücksichtigen:

$$SR(H) = \{ A, M, N, G \} \quad SR(D) = \{ A, L, B \} \quad SR(E) = \{ A, L, B \}$$
$$SR(I) = \{ M, N, G, K \} \quad SR(J) = \{ M, N, G, K \} \quad SR(C) = \{ A, B \}$$

Für zwei Mengen $SR(X)$ und $SR(Y)$ ist jetzt zu testen, ob es sich bei Referenzen, die zu

Typen aus der Schnittmenge gehören, um Referenzen auf dasselbe korrespondierende Objekt handelt.

Für die Mengen $SR(H)$ und $SR(I)$ sind damit folgende Vergleiche zu erzeugen:

$$h.m = i.m \quad h.n = i.n \quad h.g = i.g$$

Abb. 78 (S. 164) zeigt in der ersten Spalte die entstehenden Vergleiche. Allerdings sind hierin viele redundante Vergleiche enthalten. Wegen der Transitivität der $=$-Relation gilt:

$$x_1.z = x_2.z \wedge x_1.z = x_3.z \Rightarrow x_2.z = x_3.z$$

Mit $x_1 = h$, $x_2 = d$, $x_3 = e$ und $z = a$ kann daher auf den Vergleich $d.a = e.a$ verzichtet werden. Die zweite Spalte aus Abb. 78 (S. 164) zeigt die durch die Anwendung der Transitivitätsregel entstandene reduzierte Menge an Vergleichen. Auch diese Menge enthält noch redundante Elemente. Aus der Struktur des Rollenmodells folgt:

$x.z_1 = y.z_1 \wedge path(Z_2, Z_1) \Rightarrow x.z_2 = y.z_2$
mit $path(X, Y)$: es existiert im Rollenmodell ein Pfad von X nach Y

Daher kann z.B. $h.m = i.m$ gestrichen werden, weil bereits $h.g = i.g$ vorliegt und im Rollenmodell M eine Oberrolle von G ist, d.h. ein Pfad von M nach G existiert. Die Entfernung aller Vergleiche mit dieser Eigenschaft liefert die dritte Spalte in Abb. 78 (S. 164).

Als letztes läßt sich die folgende Eigenschaft ausnutzen, um die Anzahl der Vergleiche zu reduzieren:

$$x_1.z_1 = x_2.z_1 \wedge x_2.z_2 = x_3.z_2 \wedge path(Z_1, Z_2) \Rightarrow x_1.z_1 = x_3.z_1$$

Durch die Anwendung dieser Regel entsteht die zweite Spalte aus Abb. 79 (S. 165). Diese Menge besitzt keine redundanten Vergleiche mehr.

Änderungen in der `createRole`-Operation

Um die Tests für die geforderten Eigenschaften E1 bis E3 (vgl. S. 152) durchführen zu können, muß die Signatur der `createRole`-Operation um Parameter auf die benötigten Oberrollenobjekte erweitert werden. **RoleClass**$_i$ sei die direkte Unterrolle der Rolle **RoleClass**$_w$, für welche die `createRole-Operation` zu erzeugen ist. Dann müssen Referenzen für alle direkten Unterrollen **RoleClass**$_j$ von **RoleClass**$_w$ mit j$\neq$i angegeben werden, da diese für die Tests ausreichen (vgl. auch die Ausführungen zu E1 auf S. 152).

Für alle betroffenen Referenzen wird zur Kontrolle von E1 die `isDeleted`-Operation aufgerufen und gegebenenfalls eine `RoleDeleted`-Exception erzeugt. Danach wird bei allen Oberrollenobjekten kontrolliert, ob die Unterrolle aufgrund der spezifizierten Kardinalitäten überhaupt noch erzeugt werden darf (E2). Ist eine 0..1-Kardinalität verletzt, wird eine `Subrole-Exists`-Exception erzeugt, bei einer 0..n-Kardinalität (n$>$1 und n$\neq *$) eine `AllSubroles-Exist`-Exception. Als letztes sind die für E3 notwendigen Vergleichsoperationen zu erzeugen.

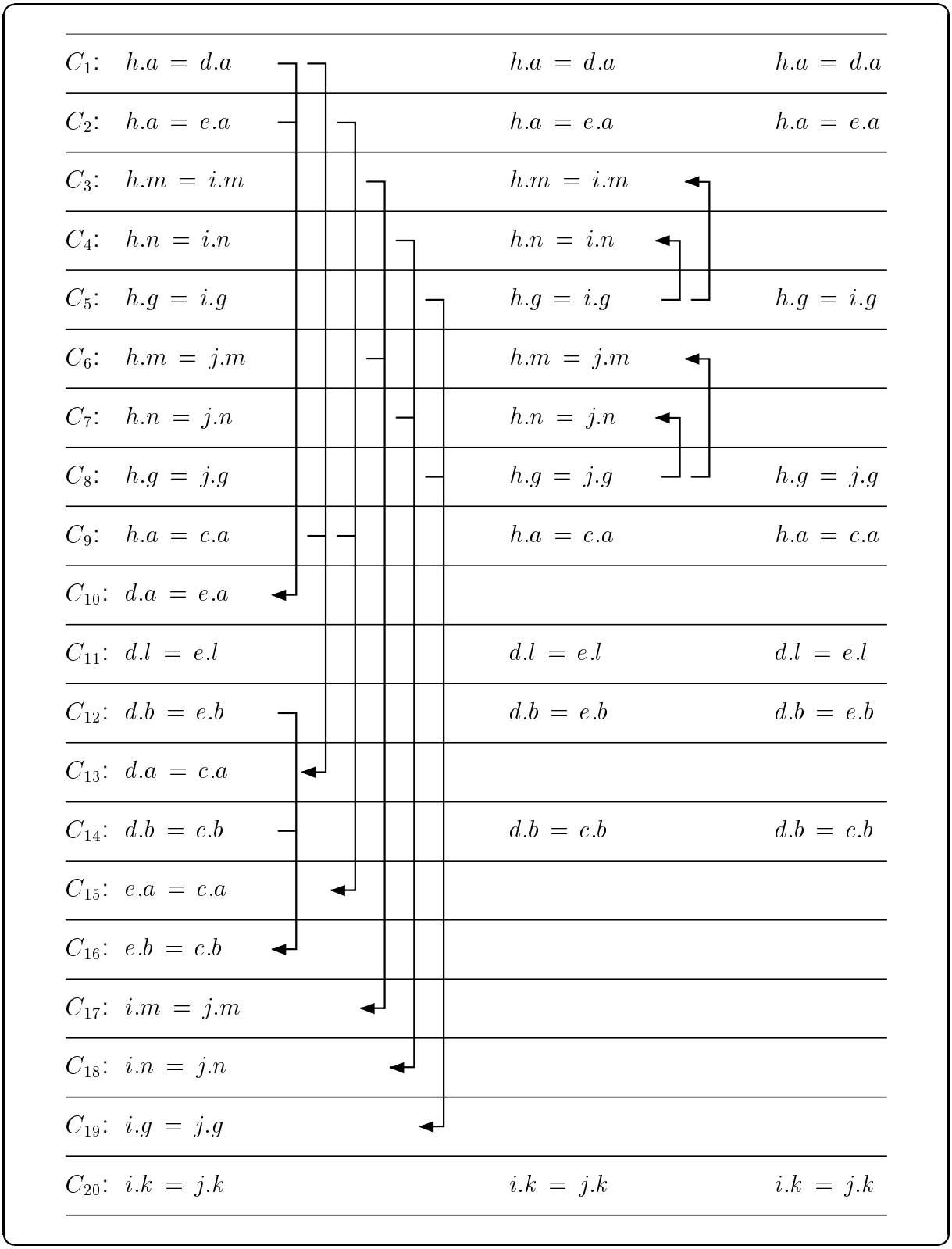


Abb. 78 Vergleiche für das Rollenmodell aus Abb. 77 (S. 162) – Teil 1

Liefert einer dieser Vergleiche ein negatives Ergebnis, wird die `createRole`-Operation mit einer `WrongSuperroles`-Exception abgebrochen.

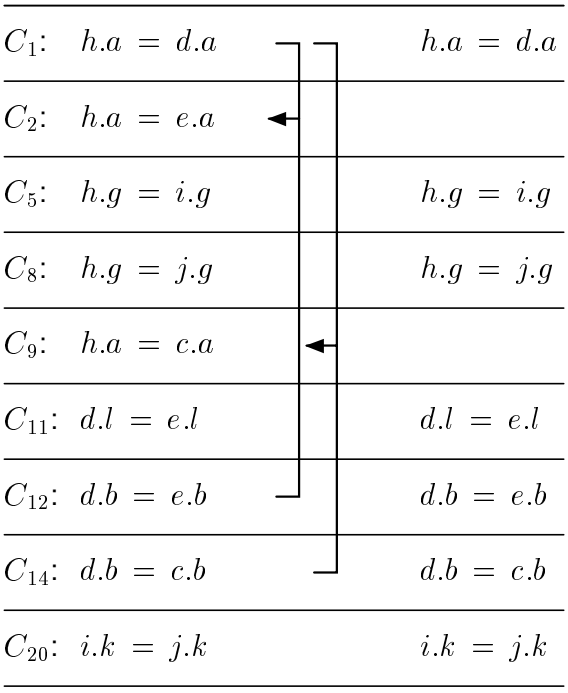


Abb. 79 Vergleiche für das Rollenmodell aus Abb. 77 (S. 162) – Teil 2

In Prg. 30 (S. 166) ist die `createRoleF`-Operation der Klasse **RoleC** für das Rollenmodell aus Abb. 77 (S. 162) dargestellt. Die Zeilen 5 bis 16 enthalten die Anweisungen zur Gewährleistung der Eigenschaft E1, die Zeilen 17 bis 28 die Anweisungen für E2, wobei für die Rollenbeziehung zwischen **J** und **F** die Kardinalität 0..3 angenommen wurde, während alle anderen Rollenbeziehungen die Kardinalität 0..1 besitzen. Daher wird in den Zeilen 27 und 28 die `isFull`-Operation verwendet und gegebenenfalls eine `AllSubrolesExist`-Exception erzeugt. Die Anweisungen zur Sicherstellung von E3 sind in den Zeilen 29 bis 42 zu sehen (vgl. auch Abb. 79).

Erzeugung der Operationen für die korrespondierenden Klassen der Oberrollen

Nur wenn keine Namenskonflikte im Rollenmodell enthalten sind, wird mit der Generierung der Rollenklassen begonnen (vgl. S. 154). Um das Konzept der Redefinition von Operationen korrekt umzusetzen, werden die korrespondierenden Klassen der Oberrollen entsprechend ihrer topologischen Sortierung verarbeitet, d.h. wenn **A**$>$**B** gilt, werden die Operationen von **A** vor denen von **B** betrachtet. Eine Operation aus **B** wird nur dann übernommen, wenn nicht bereits vorher eine Operation mit derselben Signatur generiert wurde.

In Abb. 80 (S. 167) ist eine Beispielstruktur der korrespondierenden Klassen für das Rollenmodell aus Abb. 69 (S. 153) enthalten. Die Struktur der erzeugten Rollenklassen ist in Abb. 81 (S. 168) bis Abb. 83 (S. 169) beschrieben. Die Klassen **RoleC**, **RoleD** und **RoleE** müssen die Schnittstelle **MultipleRole** implementieren, da für sie im Rollenmodell Kardinalitäten größer als 1 spezifiziert sind.

```
 1:    public RoleF createRoleF ( RoleH roleH, RoleD roleD,
 2:                RoleE roleE, RoleI roleI, RoleJ roleJ )
 3:           throws RoleDeleted, WrongSuperroles, SubroleExists,
 4:                  AllSubrolesExist {
 5:      if ( this.c==null )
 6:         throw new RoleDeleted ( "C" );
 7:      if ( roleH.isDeleted() )
 8:         throw new RoleDeleted ( "H" );
 9:      if ( roleD.isDeleted() )
10:         throw new RoleDeleted ( "D" );
11:      if ( roleE.isDeleted() )
12:         throw new RoleDeleted ( "E" );
13:      if ( roleI.isDeleted() )
14:         throw new RoleDeleted ( "I" );
15:      if ( roleJ.isDeleted() )
16:         throw new RoleDeleted ( "J" );
17:      if ( this.roleF!=null )
18:         throw new SubroleExists ( "F" );
19:      if ( roleH.roleF!=null )
20:         throw new SubroleExists ( "H/F" );
21:      if ( roleD.roleF!=null )
22:         throw new SubroleExists ( "D/F" );
23:      if ( roleE.roleF!=null )
24:         throw new SubroleExists ( "E/F" );
25:      if ( roleI.roleF!=null )
26:         throw new SubroleExists ( "I/F" );
27:      if ( roleJ.roleListF.isFull() )
28:         throw new AllSubrolesExist ( "J/F" );
29:      if ( roleH.a != roleD.a )
30:         throw new WrongSuperroles ( "H","D" );
31:      if ( roleH.g != roleI.g )
32:         throw new WrongSuperroles ( "H","I" );
33:      if ( roleH.g != roleJ.g )
34:         throw new WrongSuperroles ( "H","J" );
35:      if ( roleD.l != roleE.l )
36:         throw new WrongSuperroles ( "D","E" );
37:      if ( roleD.b != roleE.b )
38:         throw new WrongSuperroles ( "D","E" );
39:      if ( this.b != roleD.b )
40:         throw new WrongSuperroles ( "D","C" );
41:      if ( roleI.k != roleJ.k )
42:         throw new WrongSuperroles ( "I","J" );
43:      this.roleF = new RoleF ( roleH,roleD,roleE,roleI,roleJ,
44:         this );
45:      return this.roleF;
46:    }
```

Prg. 30 Die `createRoleF`-Operation der Klasse **RoleC**

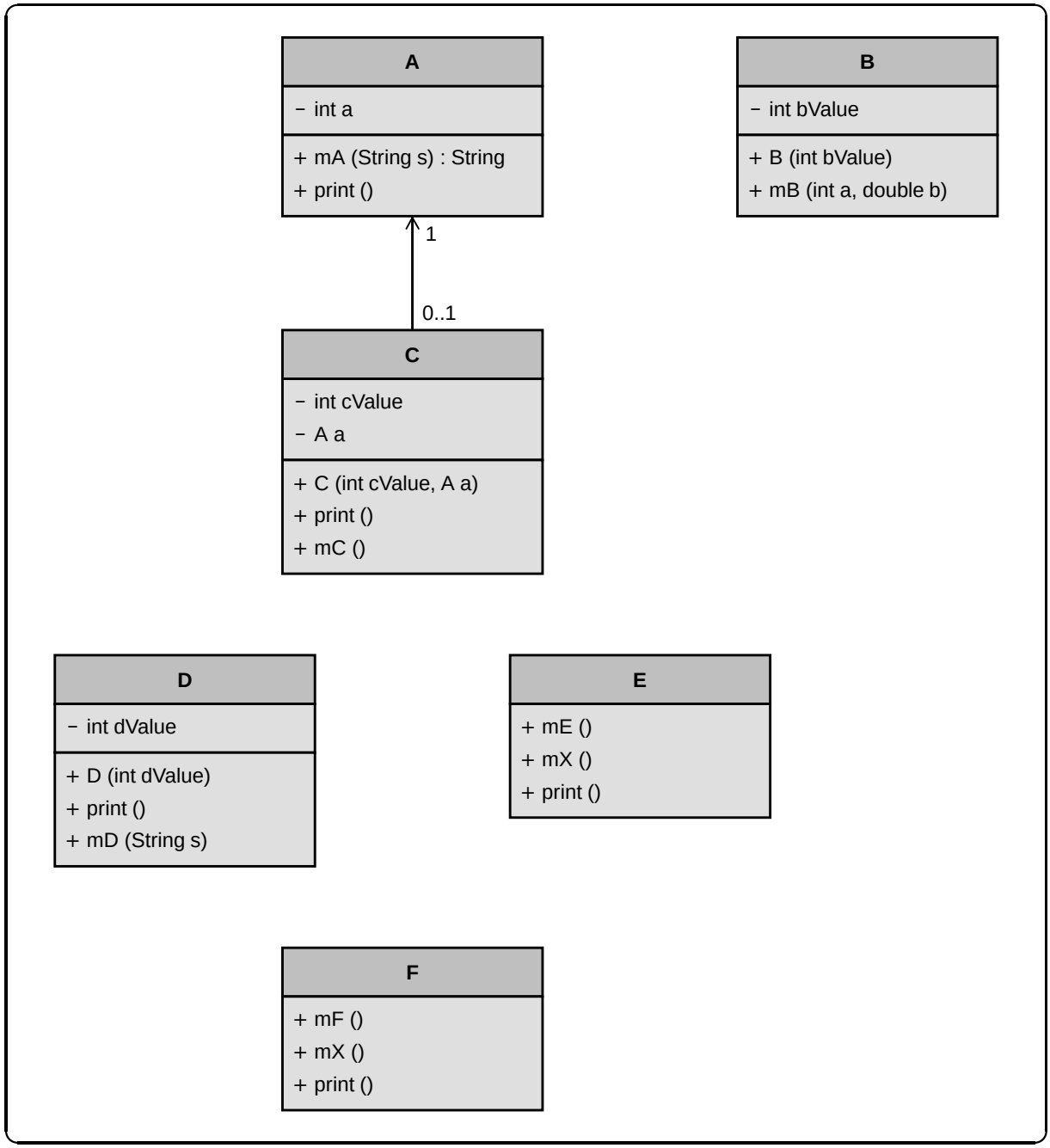


Abb. 80 Korrespondierende Klassen für das Rollenmodell aus Abb. 69 (S. 153)

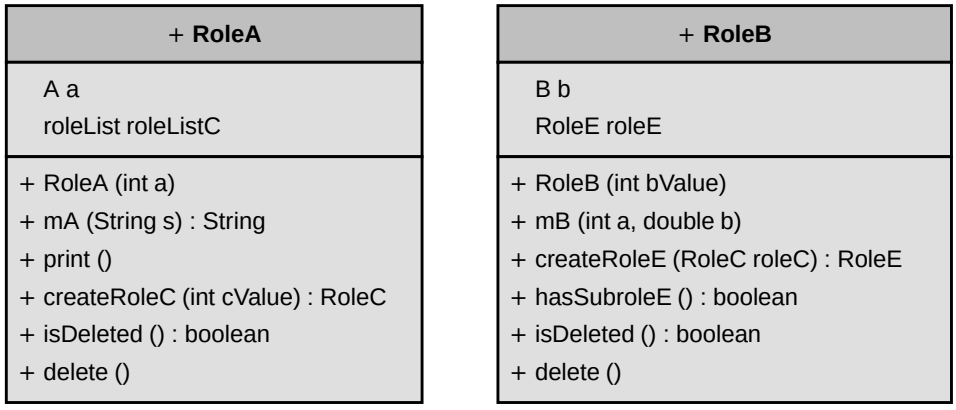


Abb. 81 Die Klassen **RoleA** und **RoleB**

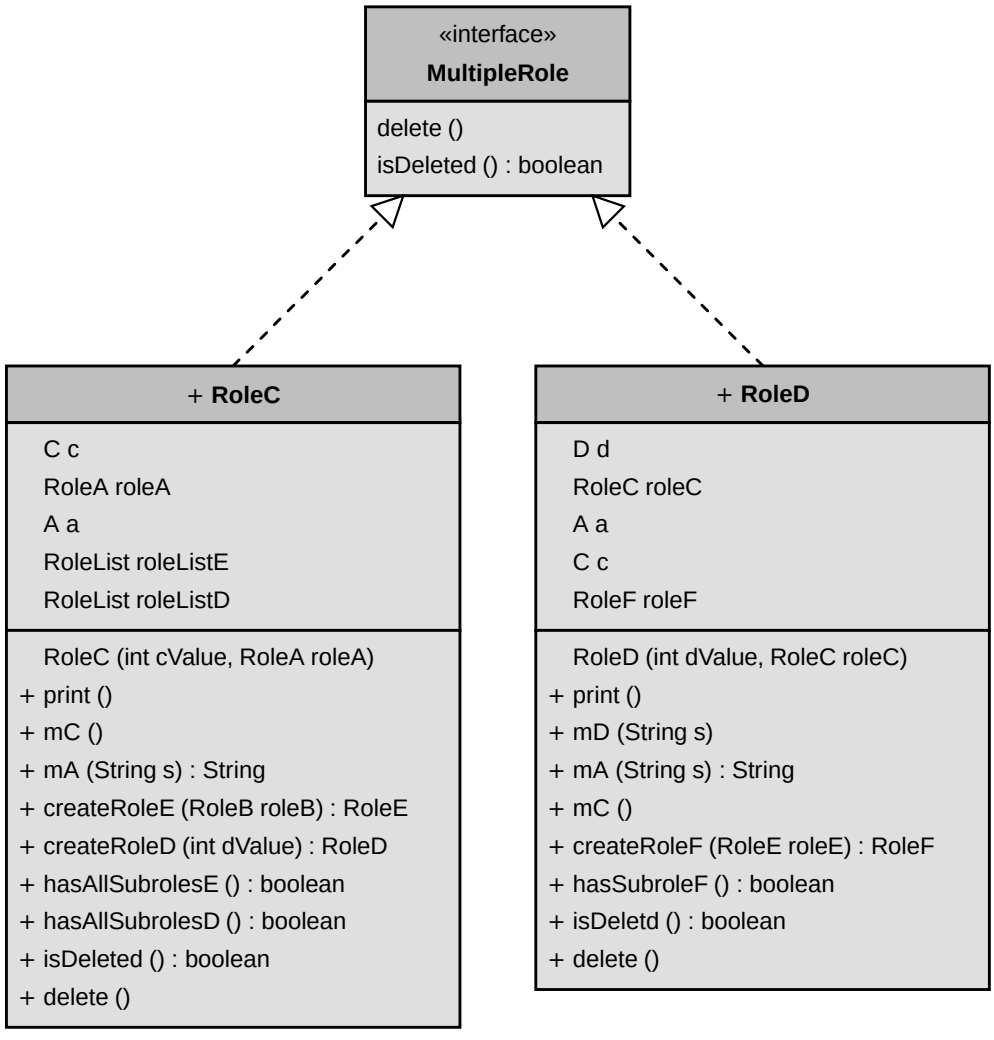


Abb. 82 Die Klassen **RoleC** und **RoleD**

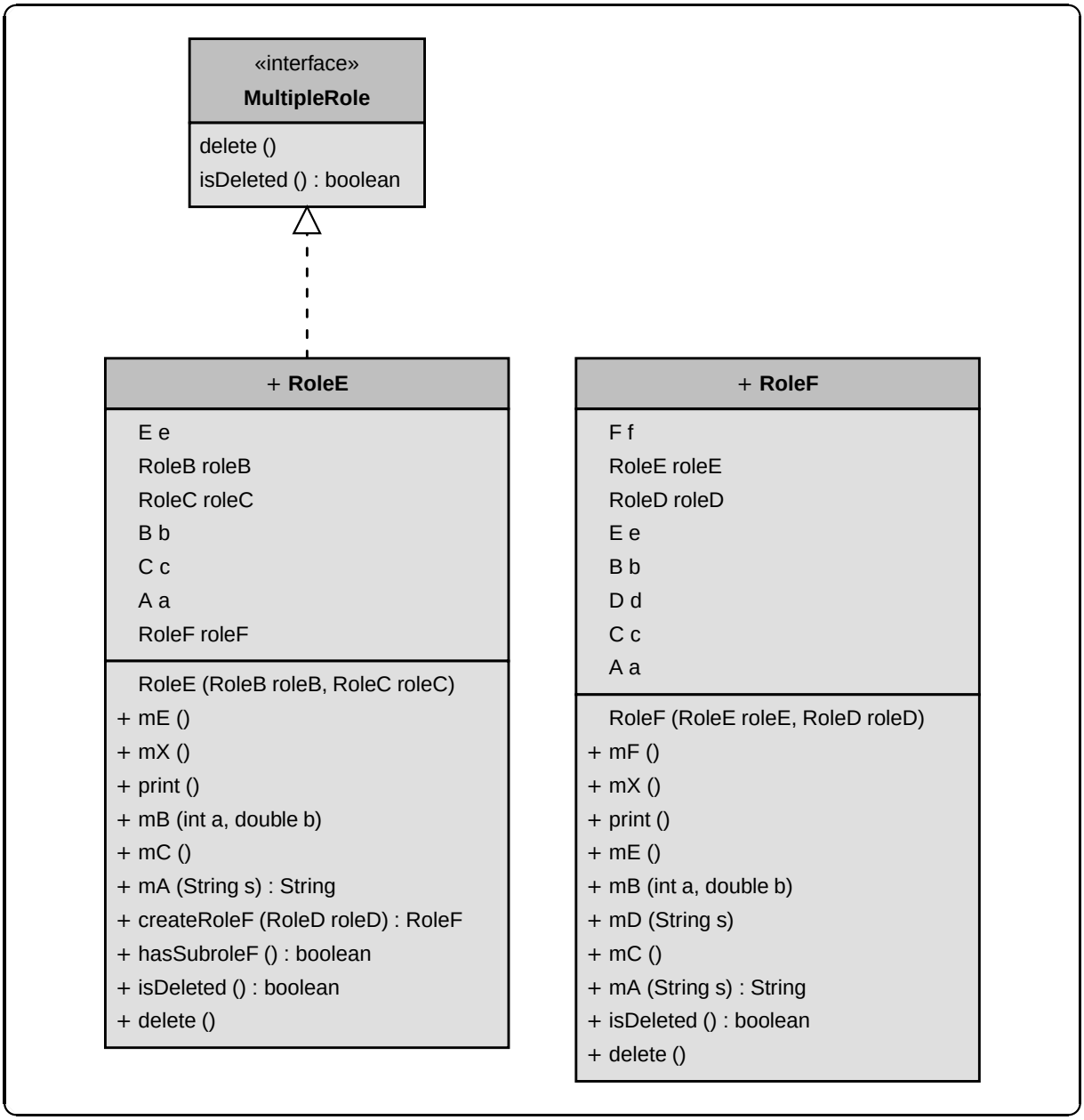


Abb. 83 Die Klassen **RoleE** und **RoleF**

Änderungen in der `delete`-Operation

Bei der `delete`-Operation ist jetzt zu berücksichtigen, daß eine Rolle mehrere direkte Oberrollen besitzen kann und daher die Referenzen auf das zu löschende Rollenobjekt überall dort ebenfalls zu löschen sind. Prg. 31 (S. 170) zeigt die `delete`-Operation der Klasse **RoleE** für das Rollenmodell aus Abb. 69 (S. 153). In den Zeilen 10 und 11 werden die Referenzen der Oberrollenobjekte auf das **RoleE**-Objekt gelöscht.

```
 1:    public void delete () throws RoleDeleted {
 2:      if ( e==null )
 3:         throw new RoleDeleted ( "E" );
 4:      e = null;
 5:      b = null;
 6:      c = null;
 7:      a = null;
 8:      if ( roleF != null )
 9:         roleF.delete();
10:      roleB.roleE = null;
11:      roleC.roleListE.remove(this);
12:      roleB = null;
13:      roleC = null;
14:    }
```

Prg. 31 Die `delete`-Operation der Klasse **RoleE**

5.4.3 Generierung des Rollenmodells

Die in Abschnitt 5.3.3 (S. 143) beschriebene Grammatik eignet sich auch zur Definition von Rollenmodellen mit Mehrfachvererbung. Vor einer Generierung des Rollenmodells wird die Konsistenz des vorliegenden Rollenmodells überprüft:

- Der über das Rollenmodell definierte Graph muß zyklenfrei sein.
- Eine als Wurzelrolle in *RootRoleRelationship* angegebene Rolle darf keine Oberrollen besitzen. Dagegen müssen alle in *RoleRelationship* enthaltenen Rollen mindestens eine Oberrolle haben, weil sie andernfalls die Eigenschaft einer Wurzelrolle erfüllen würden.
- Das Rollenmodell darf keine transitiven Rollenbeziehungen enthalten (vgl. S. 151).
- Durch die Kombination des Rollenmodells mit den korrespondierenden Klassen dürfen keine Namenskonflikte bei der Redefinition von Methoden auftreten (vgl. S. 154).

5.5 Polymorphismus

Neben der Vererbung stellt das Polymorphismuskonzept[16] eine wesentliche Grundlage zur Entwicklung wiederverwendbarer und stabiler Software-Strukturen dar. Polymorphismus steht im allgemeinen Sprachgebrauch für *Vielgestaltigkeit* bzw. *Verschiedengestaltigkeit* ([DMSW96]). Im Bereich der objektorientierten Programmierung bezieht sich dieser Begriff auf die *Fähigkeit einer Größe, zur Laufzeit auf Exemplare verschiedener Klassen zu verweisen* ([Mey90, S. 241]). Der Polymorphismus von Objekten erlaubt es, einen wiederverwendbaren Quelltext zu schreiben, der nur Referenzen auf Objekte einer Basisklasse enthält. Zur Laufzeit können diesen Referenzen dann Objekte abgeleiteter Klassen zugeordnet werden und alle Operationen auf diesen Objekten ausgeführt werden, die bereits in der Basisklasse definiert sind ([GWR00]). Die nach diesem Konzept erstellten Software-Strukturen sind damit unabhängig von den konkret vorliegenden, abgeleiteten Klassen. Wichtige Anwendungsbeispiele sind Container-Klassen, die Objektmengen verwalten, oder auch die JAVA Applet- und Servlet-Konzepte.

Das Polymorphismuskonzept wird bei der bisher vorgestellten Umsetzung von Rollenmodellen nicht unterstützt. Obwohl beispielsweise ein Student auch eine Person ist, würde die Zuweisung einer **RoleStudent**-Referenz an eine **RolePerson**-Referenz zu einem Übersetzungsfehler führen, da in der JAVA-Implementierung **RoleStudent** keine abgeleitete Klasse von **RolePerson** ist.

Bei einer Integration des Polymorphismuskonzepts ist zu berücksichtigen, daß Rollenmodelle die Mehrfachvererbung unterstützen. Da JAVA eine Mehrfachvererbung auf Klassenebene nicht vorsieht, muß eine Umsetzung auf der Grundlage der JAVA Interfaces erfolgen. Als Ausgangsbasis dienen die korrespondierenden Klassen eines Rollenmodells. Für jede Klasse **X** wird ein Interface **RoleXInt** erzeugt, das alle öffentlichen Objektmethoden von **X** enthält. Die Klassenmethoden können nicht übernommen werden, da ein JAVA Interface keine Klassenmethoden besitzen darf. Dies stellt aber keine wesentliche Einschränkung dar, weil JAVA das dynamische Binden für Klassenmethoden nicht unterstützt. Außerdem sollten Klassenmethoden nicht über eine Objektreferenz sondern über den Klassennamen aufgerufen werden. Um jetzt auf der Ebene der Interfaces die Mehrfachvererbung der Rollenklassen nachzubilden, erweitert ein Interface **RoleXInt** alle Interfaces **RoleYInt**, für die gilt, daß die zugehörige korrespondierende Klasse **Y** eine direkte Oberklasse von **X** im Rollenmodell ist. Die Interfaces, die für die Wurzelklassen des Rollenmodells erzeugt werden, enthalten zusätzlich die Operationen `delete` und `isDeleted`. Damit ist über die Vererbungsbeziehungen zwischen den Interfaces sichergestellt, daß alle Interfaces über diese beiden Methoden verfügen. Die `createRole`-, `hasAllSubrole`- und `hasSubrole`-Operationen werden dagegen nicht in die Interfaces übernommen, weil eine Unterrolle diese Operationen der Oberrolle nicht erbt.

In Abb. 84 (S. 172) sind die für das Rollenmodell aus Abb. 69 (S. 153) zu erzeugenden Interfaces zusammen mit ihren Vererbungsbeziehungen dargestellt. Die zugrunde liegenden korrespondierenden Klassen sind in Abb. 80 (S. 167) enthalten, die generierten Rollenklassen in Abb. 81 (S. 168) bis Abb. 83 (S. 169). Die Vererbungsbeziehungen der Interfaces entsprechen den Rollenbeziehungen des Rollenmodells aus Abb. 69 (S. 153). Werden innerhalb des Rollenmodells Operationen redefiniert, so treten diese Operationen mehrfach in den Interfaces

[16] Synonym zum Begriff des Polymorphismus wird auch der Begriff Polymorphie verwendet.

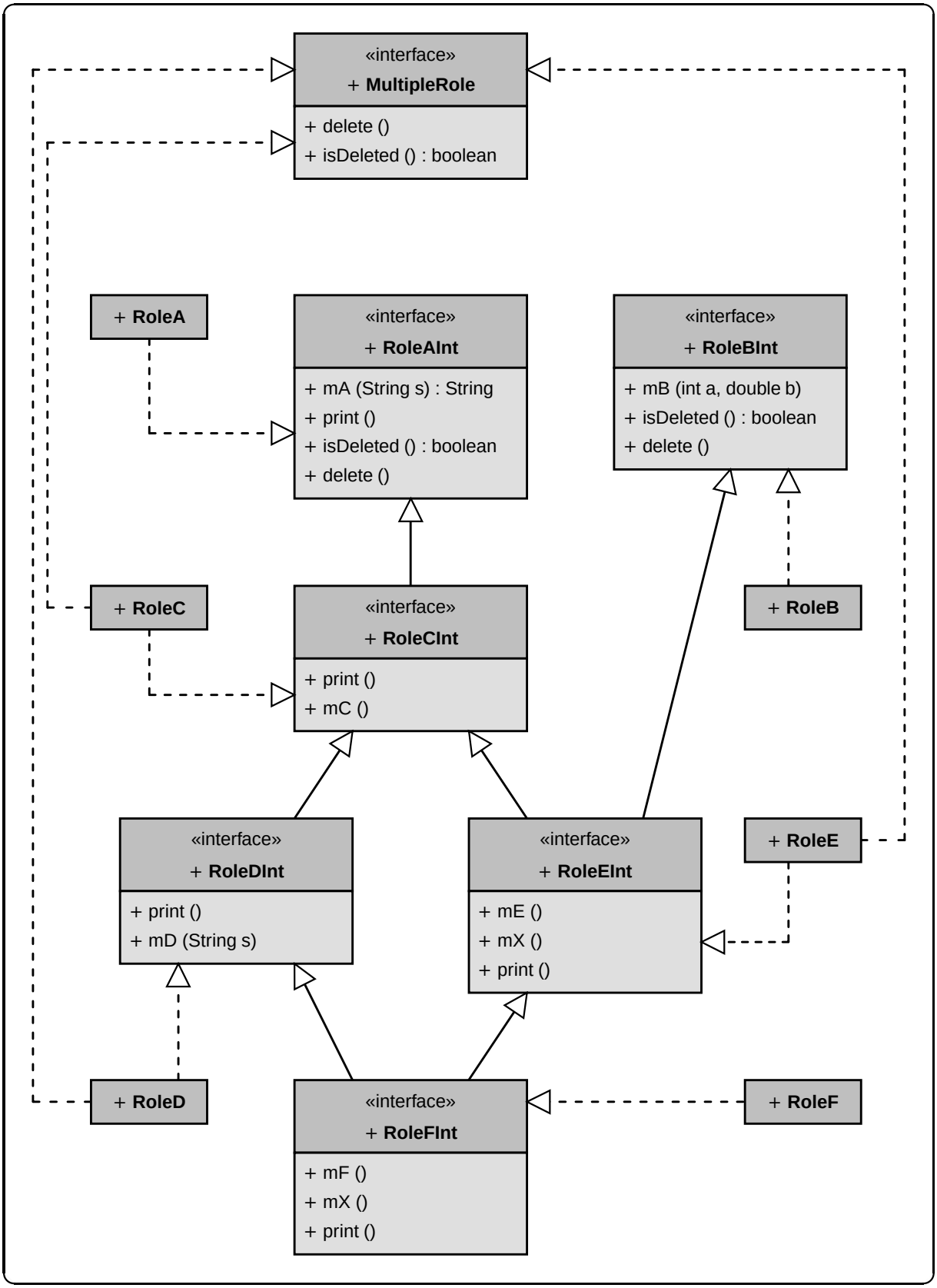


Abb. 84 Erzeugung von Interfaces für das Rollenmodell aus Abb. 69 (S. 153)

auf. Beispielsweise enthält **RoleFInt** explizit die `print`-Operation, die auch in den Interfaces **RoleEInt**, **RoleDInt**, **RoleCInt** und **RoleAInt** definiert ist. Prinzipiell könnte aufgrund der vorliegenden Vererbungsstruktur auf die Definition der `print`-Operation in den Interfaces **RoleCInt**, **RoleDInt**, **RoleEInt** und **RoleFInt** verzichtet werden. Andererseits hat die Übernahme der redefinierten Operationen den Vorteil, daß die Korrektheit der spezifizierten Exceptions beim Übersetzen der Interfaces kontrolliert wird. In JAVA gilt für die Exceptions bei redefinierten Operationen folgende Regel: wenn die redefinierte Operation eine Exception `E` enthält, die in der Exception-Liste E_1 bis E_n der Originalmethode nicht spezifiziert ist, dann muß `E` eine Unterklasse einer der Exception-Klassen E_1 bis E_n sein.

Neben der Erzeugung der Interfaces ist zur Integration des Polymorphismus-Konzepts nur noch die Aufnahme der entsprechenden **`implements`**-Klauseln in die Rollenklassen notwendig.

Prg. 32 zeigt das für die Klasse **RoleF** aus Abb. 83 (S. 169) erzeugte Interface (vgl. auch Abb. 80 (S. 167)).

```
 1: package roleModel.interfaces;
 2:
 3: import scanner.*;
 4:
 5: public interface RoleFInt
 6:        extends RoleEInt, RoleDInt {
 7:   public void mF ()
 8:          throws RoleDeleted;
 9:   public void mX ()
10:          throws RoleDeleted;
11:   public void print ()
12:          throws RoleDeleted;
13: }
```

Prg. 32 | Das Interface **RoleFInt**

5.6 Restriktionen

5.6.1 Die and- und or-Restriktionen

Restriktionen sind in der Analysephase ein wichtiges Hilfsmittel, um den zu modellierenden Auschnitt der realen Welt möglichst präzise im Analysemodell abbilden zu können. Die bisher vorgestellten Beschreibungsmechanismen für Rollenmodelle enthalten zwei Restriktionstypen:

- Da eine Unterrolle nur erzeugt werden darf, wenn für alle im Rollenmodell spezifizierten Oberrollen Rollenobjekte existieren, besteht zwischen diesen Rollen eine **and**-Restriktion.
- Für die direkten Unterrollen einer Rolle besteht dagegen eine **or**-Restriktion, weil diese Rollen unabhängig voneinander angenommen werden können.

Für das Rollenmodell aus Abb. 69 (S. 153) besteht z.B. zwischen **RoleA**, **RoleB**, **RoleC** und **RoleE** eine **and**-Restriktion; zwischen **RoleD** und **RoleE** existiert eine **or**-Restriktion.

5.6.2 Die exor-Restriktion

Betrachtet man das Rollenmodell aus Abb. 66 (S. 149), so ist aus analytischer Sicht eine **exor**-Restriktion zwischen **RoleWissenschaftlicherMitarbeiter** und **RoleVerwaltungsangestellter** sinnvoll. Um **exor**-Restriktionen für ein Rollenmodell spezifizieren zu können, wird die über *RoleModelDescriptionCA* definierte Grammatik von S. 143 entsprechend ergänzt:

```
RoleModelDescriptionRE    ::= rolemodel$ RootRoleRelationships+
                              RoleRelationships* $rolemodel
RootRoleRelationships     ::= RootRoleRelationship
                            | ExorRootRoleRelationship
RoleRelationships         ::= RoleRelationship
                            | ExorRoleRelationship
RootRoleRelationship      ::= root$ RoleIdentifier
                              hasroles$ RoleDeclaration+ $root
ExorRootRoleRelationship  ::= root$ RoleIdentifier
                              hasexorroles$ RoleDeclaration
                              RoleDeclaration+ $root
RoleRelationship          ::= role$ RoleIdentifier
                              hasroles$ RoleDeclaration+ $role
ExorRoleRelationship      ::= role$ RoleIdentifier
                              hasexorroles$ RoleDeclaration
                              RoleDeclaration+ $role
RoleDeclaration           ::= RoleIdentifier Cardinality opt
Cardinality               ::= ( ⟨ IntValue | * ⟩ )
IntValue                  ::= { 1,2,3,... }
RoleIdentifier            ::= UpperCaseLetter
                              ⟨ UpperCaseLetter | LowerCaseLetter ⟩*
UpperCaseLetter           ::= { A,B,...,Z }
LowerCaseLetter           ::= { a,b,...,z }
```

Durch die Grammatik wird sichergestellt, daß an einer **exor**-Restriktion mindestens zwei Rollen beteiligt sind. Außerdem ist es möglich, für eine Oberrolle mehrere disjunkte Unterrollenmengen zu definieren, für die jeweils die **exor**-Restriktion gilt.

In Prg. 33 ist die Beschreibung für die Graphdarstellung des Rollenmodells aus Abb. 85 enthalten, wobei die **exor**-Restriktion gemäß der UML-Notation durch {xor} dargestellt ist.

```
 1: rolemodel$
 2: root$ A hasexorroles$ C D $root
 3: root$ A hasexorroles$ E F $root
 4: root$ B hasroles$ G $root
 5: role$ C hasroles$ H $role
 6: role$ D hasroles$ H $role
 7: role$ E hasexorroles$ I J $role
 8: role$ F hasroles$ J $role
 9: role$ G hasroles$ K L $role
10: role$ H hasexorroles$ L M N $role
11: role$ J hasroles$ N $role
12: role$ K hasroles$ O $role
13: role$ L hasroles$ O $role
14: role$ M hasexorroles$ O P $role
15: $rolemodel
```

Prg. 33 Die Beschreibung des Rollenmodells aus Abb. 85

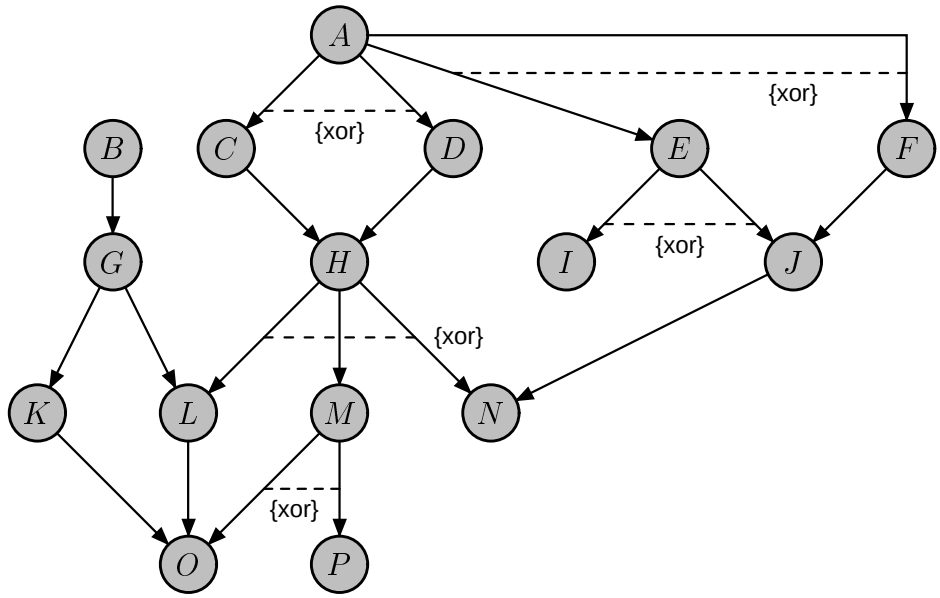


Abb. 85 Ein Rollenmodell mit **exor**-Restriktionen

Für die Rolle A sind zwei **exor**-Restriktionen definiert, die erste für die Unterrollen C und D, die zweite für die Unterrollen E und F. Ein Rollenobjekt des Typs A darf also z.B. jeweils ein Unterrollenobjekt des Typs D und F besitzen. Die Rolle H besitzt drei Unterrollen L, M und N mit einer **exor**-Restriktion. Hierbei ist zu beachten, daß in UML eine **exor**-Restriktion für eine Menge von n Elementen bedeutet, daß eine Beziehung zu maximal einem der n Elemente erlaubt ist.[17]

Durch die Integration der **exor**-Restriktion tritt in Kombination mit der Mehrfachvererbung ein weiteres Konsistenzproblem auf. Eine bestehende **exor**-Restriktion darf nicht die Erzeugung von Unterrollenobjekten verhindern, die ein anderes Unterrollenobjekt für seine Erzeugung benötigt. Das in Abb. 85 (S. 175) definierte Rollenmodell ist inkonsistent, da beispielsweise niemals ein Rollenobjekt des Typs H erzeugt werden kann. Die **exor**-Restriktion zwischen C und D verhindert die gleichzeitige Existenz entsprechender Rollenobjekte, die Voraussetzung für die Ausführung einer `createRole`-Operation für H ist. Ebenso gilt beispielsweise, daß für das Rollenmodell aus Abb. 66 (S. 149) zwischen den Rollen **RoleStudent** und **RoleUniMitarbeiter** eine **or**-Restriktion bestehen ***muß***, da andernfalls ein Student keine Tutorentätigkeit übernehmen könnte.

Aus dieser Forderung läßt sich ein allgemeines Konsistenzkriterium für die Verwendung von **exor**-Restriktionen bei Rollenmodellen mit Mehrfachvererbung ableiten. Damit eine Unterrolle u überhaupt erzeugbar ist, muß die folgende Eigenschaft erfüllt sein: innerhalb der Menge der Oberrollen $SPR(u)$ von u darf zwischen zwei beliebigen Oberrollen x und y keine **exor**-Restriktion existieren. Zur Kontrolle dieser Eigenschaft kann der folgende Test verwendet werden:

$$\forall x \in NRSPR(u) : EXOR(x) \cap NRSPR(u) = \emptyset \quad \text{mit}$$

$NRSPR(u) \stackrel{def}{=}$ Menge aller Oberrollen von u, die keine Wurzelrollen sind

$$= SPR(u) \setminus RR(RM)$$

$SPR(u) \stackrel{def}{=}$ Menge aller Oberrollen von u

$RR(RM) \stackrel{def}{=}$ Menge aller Wurzelrollen des Rollenmodells RM

$EXOR(x) \stackrel{def}{=}$ Menge aller Rollen, die in einer **exor**-Beziehung zu x stehen

Bei dem Test kann – wie oben definiert – die Menge $NRSPR(u)$ statt der Menge $SPR(u)$ verwendet werden, weil die Wurzelrollen eines Rollenmodells immer erzeugbar sind. Für das Rollenmodell aus Abb. 85 (S. 175) gilt z.B. für $u = P$:

$$RR(RM) = \{ B, A \}$$
$$SPR(P) = \{ M, H, C, D, A \}$$
$$NRSPR(P) = SPR(N) \setminus RR(RM) = \{ M, H, C, D \}$$
$$EXOR(M) = \{ L, N \} \quad EXOR(H) = \emptyset \quad EXOR(C) = \{ D \} \quad EXOR(D) = \{ C \}$$
$$NRSPRP \cap EXOR(M) = \emptyset \quad NRSPR(P) \cap EXOR(H) = \emptyset$$
$$NRSPR(P) \cap EXOR(C) = \{ D \} \quad NRSPR(P) \cap EXOR(C) = \{ D \}$$

[17] Im Gegensatz zur UML-Definition liefert z.B. der boolesche Ausdruck $(a\ exor\ b)\ exor\ c$ auch den Wert $true$, wenn a, b und c jeweils $true$ sind.

Das Rollenmodell ist inkosistent, da z.B. $NRSPR(P) \cap EXOR(C) \neq \emptyset$ gilt.

Betrachtet man das Beispiel aus Abb. 85 (S. 175), so könnte man vermuten, daß es zur Beurteilung der Erzeugbarkeit einer Unterrolle jeweils ausreicht zu überprüfen, daß keine **exor**-Beziehung zwischen den direkten Oberrollen besteht. Diese Vermutung ist jedoch falsch, wie das folgende Gegenbeispiel aus Abb. 86 zeigt.

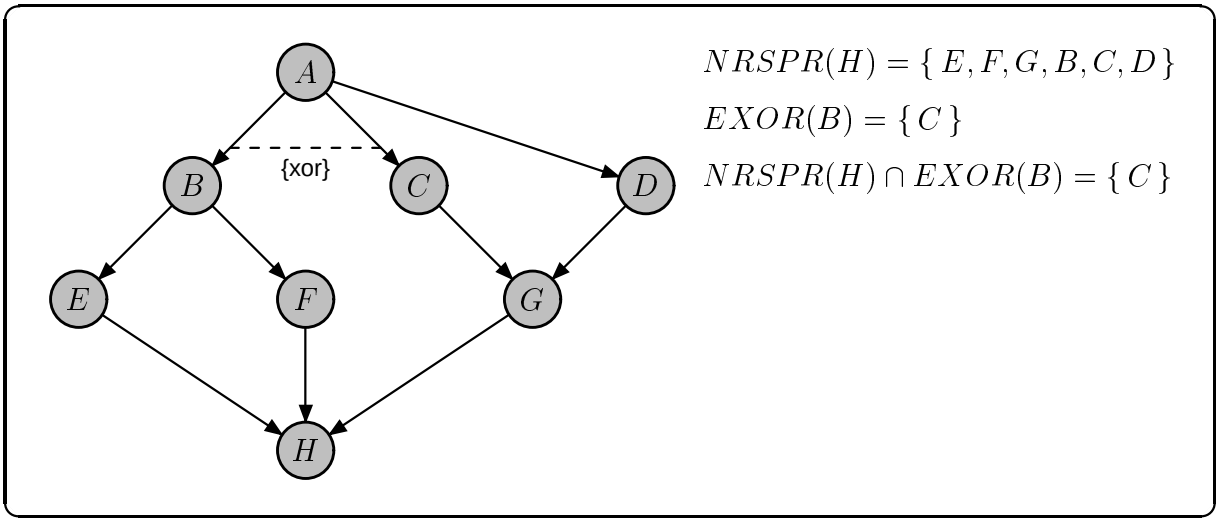


Abb. 86 Konstruktion eines Gegenbeispiels

Die direkten Oberrollen von H stehen in keiner **exor**-Beziehung zueinander, die direkte Oberrolle C von G hat zwar eine **exor**-Beziehung zu B, was die Erzeugung von G aber nicht betrifft. Trotzdem ist das Rollenmodell wegen $NRSPR(H) \cap EXOR(B) \neq \emptyset$ inkonsistent. Jede der direkten Oberrollen E, F und G von H ist zwar einzeln erzeugbar, ihre Kombination kann aber gleichzeitig nicht vorliegen.

Um die Konsistenz des Rollenmodells zu überprüfen, reicht es aus, den auf S. 176 beschriebenen Test für alle Rollen auszuführen, die selbst keine Unterrollen mehr haben. Diese Menge soll für ein Rollenmodell RM mit $LR(RM)$ bezeichnet werden, und entspricht genau den Blattknoten der Graphdarstellung des Rollenmodells. Die obige Aussage läßt sich aus den beiden folgenden Eigenschaften ableiten. Zum einen bedeutet ein erfolgreicher Test für eine Rolle x, daß auch alle Oberrollen erzeugbar sein müssen, andernfalls wäre x nicht erzeugbar gewesen und damit der Test fehlgeschlagen. Zum anderen enthält die Vereinigungsmenge aller $NRSPR(x)$-Mengen mit $x \in LR(RM)$ alle Rollen des Rollenmodells mit Ausnahme der Wurzelrollen, für die der Test aber nicht notwendig ist.

Der Test für eine Rolle x aus $LR(RM)$ enthält aufgrund der Kommutativität der $exor$-Operation einige redundante Schnittmengenbildungen. Auf eine weitere Optimierung wird an dieser Stelle jedoch verzichtet, da die Konsistenz des Rollenmodells bezüglich der **exor**-Restriktionen nur einmal vor der Generierung der Rollenklassen durchzuführen ist.

Zusätzlich zu dem Konsistenztest ist zur Integration der **exor**-Restriktionen in ein Rollenmodell eine Änderung der `createRole`-Operationen notwendig. Vor der Erzeugung eines Unterrollenobjekts ist zu prüfen, ob nicht bereits ein Unterrollenobjekt aus der zugehörigen $EXOR$-Menge existiert.

In Prg. 34 ist die `createRoleD`-Operation aus **RoleA** für das Rollenmodell aus Abb. 87 dargestellt. In den Zeilen 7 bis 9 ist der Test enthalten, ob bereits andere Unterrollenobjekte aus den Klassen **RoleE** oder **RoleF** existieren. Hierbei ist zu beachten, daß aufgrund der spezifizierten Kardinalitäten unterschiedliche Bedingungen generiert wurden.

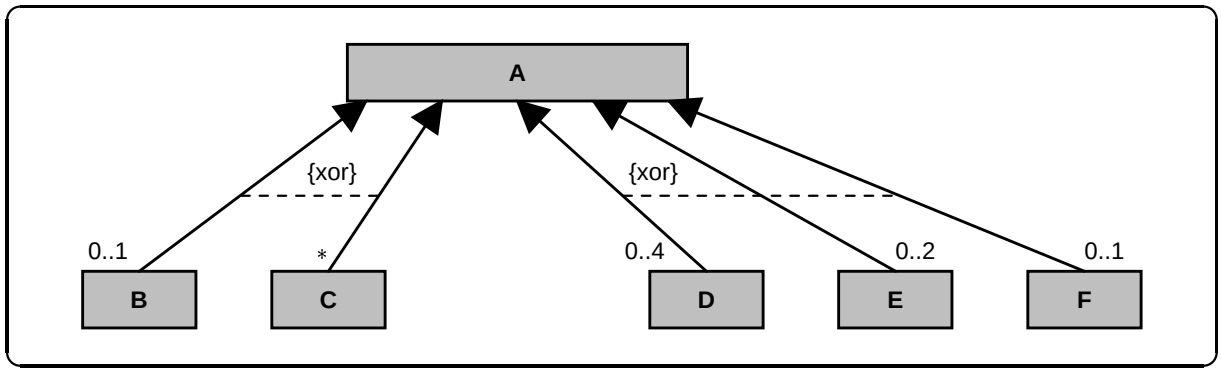


Abb. 87 Ein Rollenmodell mit **exor**-Restriktionen

```
 1:    public RoleD createRoleD ()
 2:           throws AllSubrolesExist,ExorSubroleExists,RoleDeleted {
 3:      if ( a==null )
 4:         throw new RoleDeleted ( "A" );
 5:      if ( roleListD.isFull() )
 6:         throw new AllSubrolesExist ( "D" );
 7:      if ( ! this.roleListE.isEmpty() ||
 8:           this.roleF!=null )
 9:         throw new ExorSubroleExists();
10:      RoleD roleD;
11:      roleD = new RoleD ( this,a );
12:      roleListD.insert(roleD);
13:      return roleD;
14:    }
```

Prg. 34 Die `createRoleD`-Operation der Klasse **RoleA** für das Rollenmodell aus Abb. 87

5.6.3 Die nocreation-Restriktion

In Abschnitt 5.6.2 (S. 174) wurde das Rollenmodell einer einfachen Universitätsverwaltung aus Abb. 66 (S. 149) bereits um eine **exor**-Restriktion erweitert, um die Realität genauer abzubilden. Eine weitere Anforderung auf der Analyseebene wird sein, daß nur Studenten Tutoren werden dürfen. In dem bisher spezifizierten Universitätsmodell wäre es möglich, daß ein Universitätsmitarbeiter, der die Rolle **Verwaltungsmitarbeiter** besitzt, eine Tutorenrolle übernimmt. Um diesen Fall auszuschließen, wird die Rollenbeziehung zwischen **UniMitarbeiter** und **Tutor** mit der **nocreation**-Restriktion versehen. Diese sagt aus, daß ausgehend von einem **RoleUniMitarbeiter**-Objekt kein **RoleTutor**-Objekt erzeugt werden kann. Abb. 88 enthält die Definition des zugehörigen Rollenmodells.

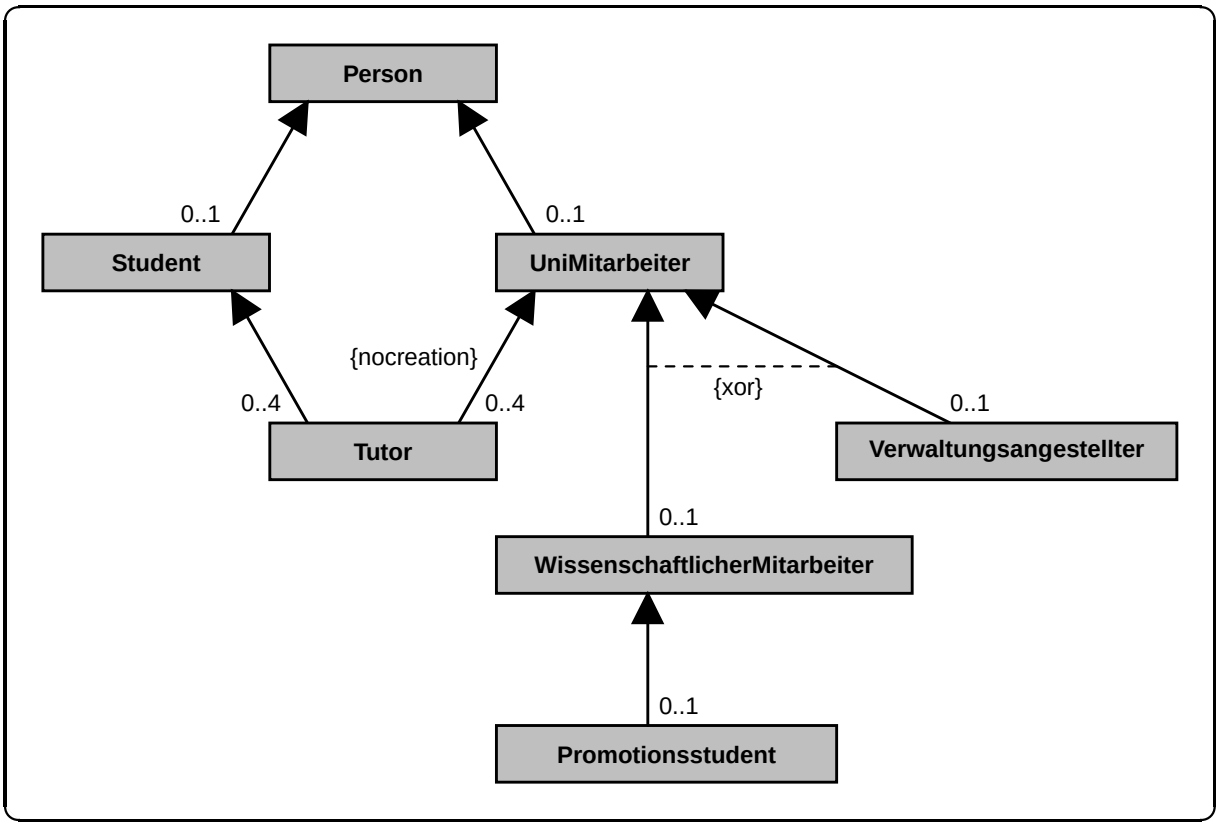


Abb. 88 Ein Rollenmodell mit der **nocreation**-Restriktion

Für die Berücksichtigung von **nocreation**-Restriktionen müssen in der Grammatik aus Abschnitt 5.6.2 (S. 174) nur die *RoleDeclaration*-Produktion geändert und eine *NoCreation*-Produktion ergänzt werden:

RoleDeclaration ::= *RoleIdentifier* *NoCreation*^{opt} *Cardinality*^{opt}
NoCreation ::= **[nocreation$]**

Die Strukturbeschreibung des Rollenmodells aus Abb. 88 ist in Prg. 35 (S. 180) dargestellt.

Wie bei der **exor**-Restriktion existiert auch für die **nocreation**-Restriktion ein Konsistenzkriterium für das Rollenmodell. Es darf nicht passieren, daß durch die Spezifikation der **nocreation**-

```
 1: rolemodel$
 2: root$ Person hasroles$ Student UniMitarbeiter $root
 3: role$ Student hasroles$ Tutor(4) $role
 4: role$ UniMitarbeiter
 5:       hasroles$ Tutor[nocreation$](4) $role
 6: role$ UniMitarbeiter hasexorroles$
 7:                      Verwaltungsangestellter
 8:                      WissenschaftlicherMitarbeiter
 9: $role
10: role$ WissenschaftlicherMitarbeiter
11:       hasroles$ Promotionsstudent $role
12: $rolemodel
```

Prg. 35 Die Beschreibung des Rollenmodells aus Abb. 88 (S. 179)

Restriktionen Unterrollen nicht mehr erzeugbar sind. Betrachtet man den Graphen des Rollenmodells, so muß sichergestellt sein, daß nach einer Entfernung aller Kanten, für die eine **nocreation**-Restriktion definiert wurde, der Graph nicht in Teilgraphen zerfällt. In Abb. 89 ist ein inkonsistentes Rollenmodell dargestellt, da nach Entfernung der **nocreation**-Kanten aus dem linken Graphen vier Teilgraphen entstehen.

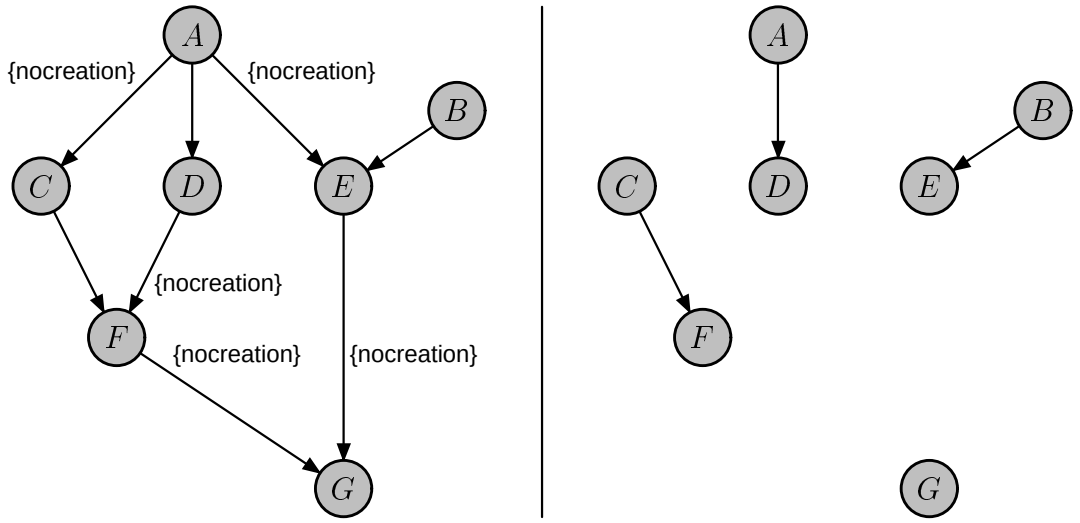


Abb. 89 Ein Rollenmodell mit inkonsistenten **nocreation**-Restriktionen

Der notwendige Test ist relativ einfach aufgebaut. Es ist ausreichend zu kontrollieren, daß zu jedem Nichtwurzelknoten des Graphen mindestens eine Kante ohne **nocreation**-Restriktion führt. Diese hinreichende Bedingung folgt aus der Eigenschaft eines Rollenmodells, daß der zugehörige Graph zyklenfrei ist.

Auf der Entwurfsebene wird zur Umsetzung einer **nocreation**-Restriktion zwischen den Rollen **RoleA** und **RoleB** die `createRoleB`-Operation in **RoleA** einfach weggelassen. Damit ist es nicht mehr möglich, auf diesem Weg **RoleB**-Objekte zu erzeugen.

5.7 Abstrakte Rollen

Abstrakte Klassen spielen in allen Phasen des Software-Entwicklungsprozesses eine wichtige Rolle. In der Analysephase sind sie häufig das Ergebnis eines Generalisierungsprozesses zur Konstruktion von Vererbungshierarchien. Sie werden immer dann eingesetzt, wenn eine Oberklasse entsteht, von der selbst aber keine Objekte erzeugt werden sollen. Beispielsweise könnte eine Klassenhierarchie zur Modellierung geometrischer Objekte eine abstrakte Oberklasse **GeometrischesObjekt** besitzen. Von den abgeleiteten Klassen, wie z.B. **Rechteck**, **Dreieck** und **Kreis**, können dann konkrete Objekte erzeugt werden. Auf der Entwurfsebene werden abstrakte Klassen oft zusammen mit dem Polymorphismus-Konzept eingesetzt. Hier sind Container-Klassen ein wichtiges Beispiel. Will man z.B. für einen grafischen Editor alle erzeugten geometrischen Objekte verwalten, so bietet sich eine Listen- oder Baumstruktur an, die über der abstrakten Klasse **GeometrischesObjekt** definiert ist. Alle Operationen aus der abstrakten Klasse können dann auf den konkret in der Container-Struktur enthaltenen Objekten ausgeführt werden, wobei über das dynamische Binden die entsprechenden Operationen der Objekte Verwendung finden.

Auch für die hier vorgestellte Struktur eines Rollenmodells ist es sinnvoll, zwischen abstrakten und konkreten Rollen zu differenzieren. Betrachtet man z.B. das Rollenmodell einer Universitätsverwaltung (vgl. Abb. 88 (S. 179)), so stellen **RoleStudent**, **RoleTutor**, **RoleWissenschaftlicherMitarbeiter**, **RoleVerwaltungsangestellter** und **RolePromotionsstudent** sicherlich konkrete Rollen dar. Bei den Rollen **RolePerson** und **RoleUniMitarbeiter** ist innerhalb der Analysephase zu überlegen, ob sie überhaupt als eigenständige Rollen existieren sollen oder nur in Kombination mit den entsprechenden konkreten Unterrollen, wie z.B. **RoleTutor**.

Für die Intergration abstrakter Rollen wird die Grammatik *RoleModelDescriptionRE* aus Abschnitt 5.6.2 (S. 174) erweitert, wobei hier auch die Änderungen zur Unterstützung der **no-creation**-Restriktion von S. 179 eingearbeitet sind:

```
RoleModelDescription      ::= rolemodel$ RootRoleRelationships+
                              RoleRelationships* $rolemodel
RootRoleRelationships     ::= RootRoleRelationship
                            | ExorRootRoleRelationship
RoleRelationships         ::= RoleRelationship
                            | ExorRoleRelationship
RootRoleRelationship      ::= root$ RoleDeclaration1
                              hasroles$ RoleDeclaration2+ $root
ExorRootRoleRelationship  ::= root$ RoleDeclaration1
                              hasexorroles$ RoleDeclaration2
                              RoleDeclaration2+ $root
RoleRelationship          ::= role$ RoleDeclaration1
                              hasroles$ RoleDeclaration2+ $role
ExorRoleRelationship      ::= role$ RoleDeclaration1
                              hasexorroles$ RoleDeclaration2
                              RoleDeclaration2+ $role
RoleDeclaration1          ::= abstractopt RoleIdentifier
RoleDeclaration2          ::= RoleIdentifier NoCreationopt
```

```
                                Cardinality^opt
NoCreation              ::= [nocreation$]
Cardinality             ::= ( ⟨ IntValue | * ⟩ )
IntValue                ::= { 1,2,3,... }
RoleIdentifier          ::= UpperCaseLetter
                            ⟨ UpperCaseLetter | LowerCaseLetter ⟩*
UpperCaseLetter         ::= { A,B,...,Z }
LowerCaseLetter         ::= { a,b,...,z }
```

Durch die Struktur der Grammatik wird sichergestellt, daß eine abstrakte Rolle auf jeden Fall eine konkrete Unterrolle besitzen muß, da das Schlüsselwort **`abstract`** nur in *RoleDeclaration1* auftreten darf.

Für die Umsetzung abstrakter Rollen wird auf der Entwurfsebene der folgende Ansatz gewählt:

- Für eine abstrakte Rolle **RoleA** werden nur noch die Verwaltungsoperationen generiert. Die Operationen aus der korrespondierenden Klasse und den korrespondierenden Klassen der Oberrollen treten in **RoleA** nicht mehr auf.
- Das zugehörige Interface **RoleAInt** enthält aber weiterhin die Signaturen aller öffentlichen Operationen aus **A**, die Rollenklasse **RoleA** implementiert das Interface jedoch nicht mehr.

Diese Vorgehensweise ermöglicht es, auch von abstrakten Rollen Objekte zu erzeugen. Diese werden von den Unterrollenobjekten benötigt, da eine Unterrolle alle Eigenschaften ihrer Oberrollen erbt, unabhängig davon, ob es sich um konkrete oder abstrakte Rollen handelt.

Der wesentliche Unterschied zu abstrakten Klassen besteht darin, daß die in einer abstrakten Klasse definierten Attribute nur dann erzeugt werden[18], wenn ein Objekt einer Unterklasse angelegt wird. Diese Attribute können daher nicht unabhängig von einem Unterklassenobjekt existieren. Bei dem hier vorgestellten Ansatz darf dagegen ein Objekt einer abstrakten Rolle existieren, ohne daß bereits ein Unterrollenobjekt einer nicht abstrakten Rollenklasse vorhanden ist. Dieses Objekt kann aber aus der Sicht der Fachkonzeptschicht als abstrakt angesehen werden, da keine der zum Fachkonzept gehörenden Operationen[19] über die entsprechende Rollenobjektreferenz aufrufbar ist.

Das für eine abstrakte Rolle erzeugte Interface dient zur Umsetzung der Polymorphismuskonzepts. Will man beispielsweise alle Universitätsmitarbeiter verwalten, bietet sich in JAVA eine Array- oder Listenstruktur an. Benutzt man ein Array, so wird dieses unter Verwendung des Interfaces **RoleUniMitarbeiterInt** definiert. Es kann Rollenobjektreferenzen aller Unterrollen von **RoleUniMitarbeiter** aufnehmen, wobei auf den Rollenobjekten dann alle in der korrespondierenden Klasse **UniMitarbeiter** definierten öffentlichen Operationen aufrufbar sind.

In dem folgenden Beispiel wird eine Klasse **Mitarbeiterverwaltung** beschrieben, die alle diejenigen Personen einer Universität aufnehmen kann, die die abstrakte Rolle **RoleUniMitarbeiter** besitzen. Als Grundlage dienen das Rollenmodell aus Abb. 88 (S. 179) sowie die korrespondierenden Klassen aus Abb. 90 (S. 183). Jede dieser Klassen besitzt eine Ausgabeoperation, welche die für die zugehörige Rolle neu hinzugekommenen Daten ausgibt (z.B. `printStudent` in **Student**), sowie eine `print`-Operation, die die vollständigen Daten eines Rollenobjekts

[18] Mit der Attributerzeugung ist hier die Speicherplatzreservierung und die Initialisierung gemeint.

[19] Dieses sind genau die öffentlichen Operationen der korrespondierenden Klasse und der korrespondierenden Klassen der Oberrollen.

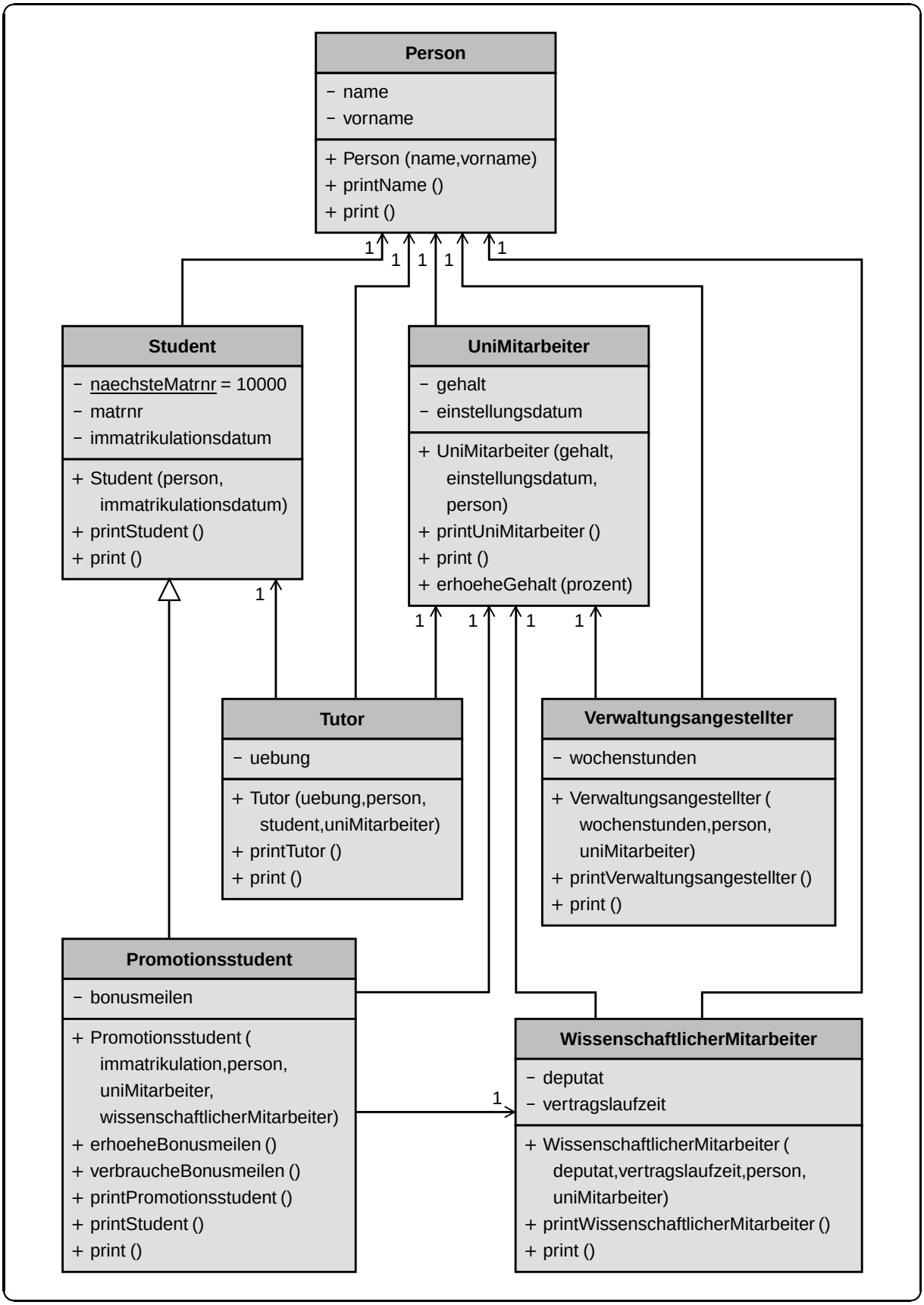


Abb. 90 Die Struktur der korrespondierenden Klassen des Modells aus Abb. 88 (S. 179)

liefert, d.h. auch die Daten aller Oberrollenobjekte. Deshalb besitzt ein Objekt einer korrespondierenden Klasse **X** Beziehungen zu den Objekten von denjenigen korrespondierenden Klassen **Y**, die im Rollenmodell die Struktur der Oberrollen von **RoleX** festlegen. Diese Beziehungen werden dann innerhalb der `print`-Operationen genutzt, um auf die entsprechenden `printY`-Operationen zuzugreifen. In Prg. 36 ist die korrespondierende Klasse **Tutor** dargestellt, deren `print`-Operation auf die Beziehungen zu den **Student**-, **UniMitarbeiter**- und **Person**-Objekten zurückgreift.

```
 1: package roleModel.abstractRoles;
 2:
 3: class Tutor {
 4:   private String uebung;
 5:   private Person person;
 6:   private Student student;
 7:   private UniMitarbeiter uniMitarbeiter;
 8:
 9:   public Tutor ( String uebung, Person person,
10:                  Student student,
11:                  UniMitarbeiter uniMitarbeiter ) {
12:     this.uebung = uebung;
13:     this.person = person;
14:     this.student = student;
15:     this.uniMitarbeiter = uniMitarbeiter;
16:   }
17:
18:   public void printTutor() {
19:     System.out.println ( "Tutor.print: " +
20:       "betreute Lehrveranstaltung=" + uebung );
21:   }
22:
23:   public void print () {
24:     person.print();
25:     student.printStudent();
26:     uniMitarbeiter.printUniMitarbeiter();
27:     printTutor();
28:   }
29: }
```

Prg. 36 Der Quelltext der korrespondierenden Klasse **Tutor**

Die Klasse **Promotionsstudent** wurde als Erweiterung von **Student** definiert, da ein Promotionsstudent alle Eigenschaften eines Studenten besitzt. Auf der Ebene des Rollenmodells stehen die beiden Rollen **RoleStudent** und **RolePromotionsstudent** dagegen in keiner Rollenbeziehung. In einer alternativen Modellierung hätte **RolePromotionsstudent** die direkten Oberrollen **RoleWissenschaftlicherMitarbeiter** und **RoleStudent** haben können. Dann wäre es sinnvoll gewesen, bei der Definition des Rollenmodells eine **nocreation**-Restriktion zwischen **Student** und **Tutor** zu spezifizieren, da nur ein wissenschaftlicher Mitarbeiter diese Rolle über-

nehmen darf und nicht ein beliebiger Student.

Die Abbildungen Abb. 91 bis Abb. 97 (S. 188) zeigen die jeweilige Struktur der zu generierenden Rollenklassen.

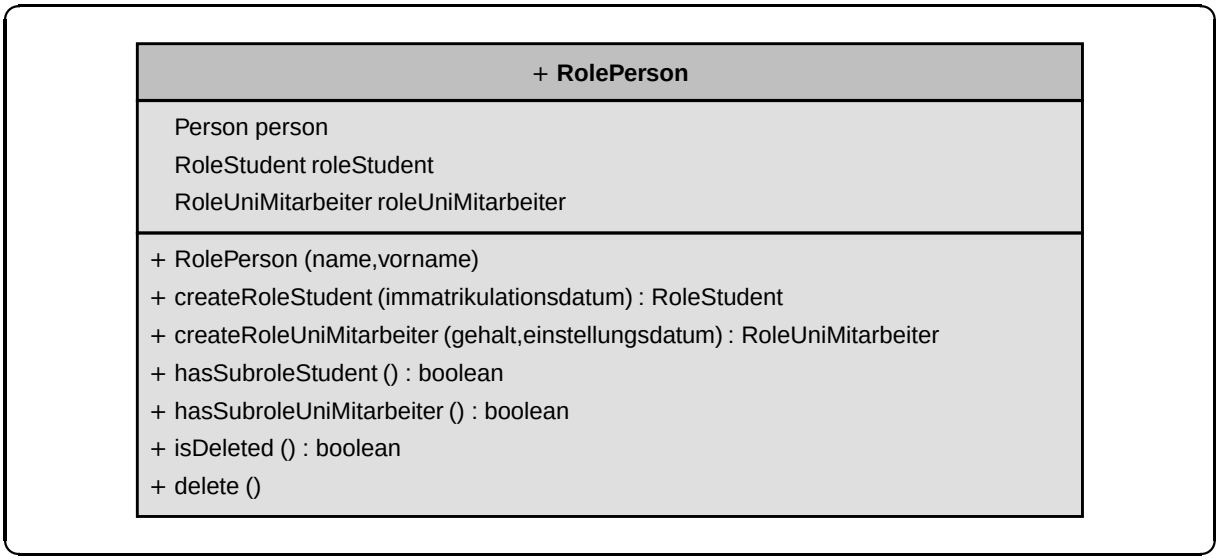


Abb. 91 Die Struktur der Klasse **RolePerson**

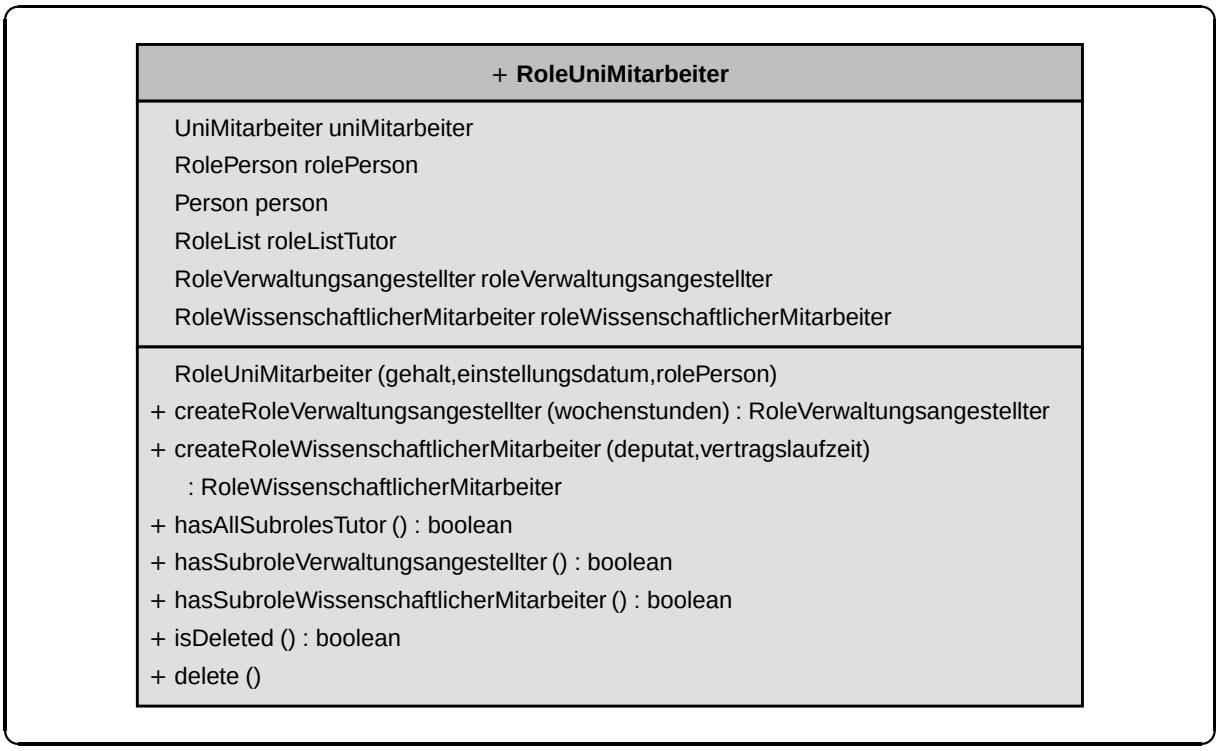


Abb. 92 Die Struktur der Klasse **RoleUniMitarbeiter**

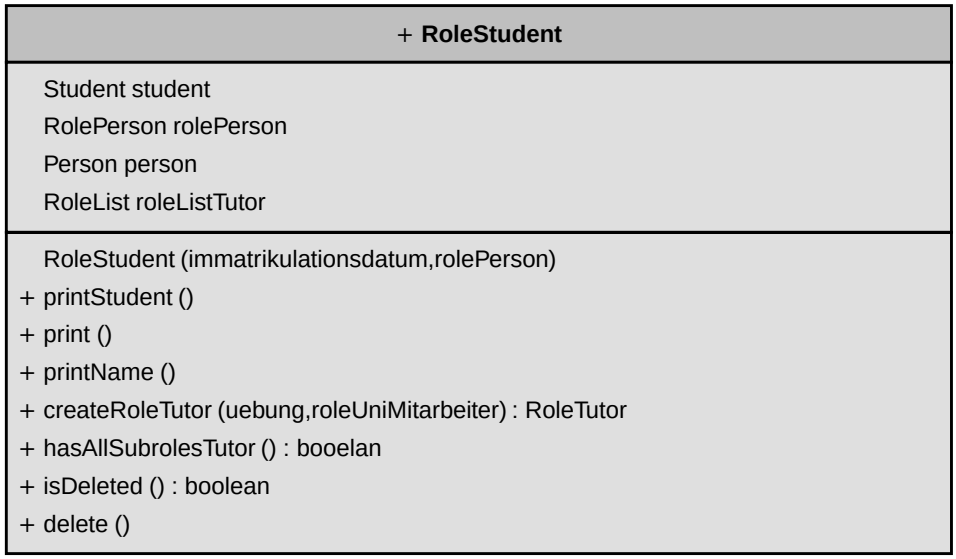


Abb. 93 Die Struktur der Klasse **RoleStudent**

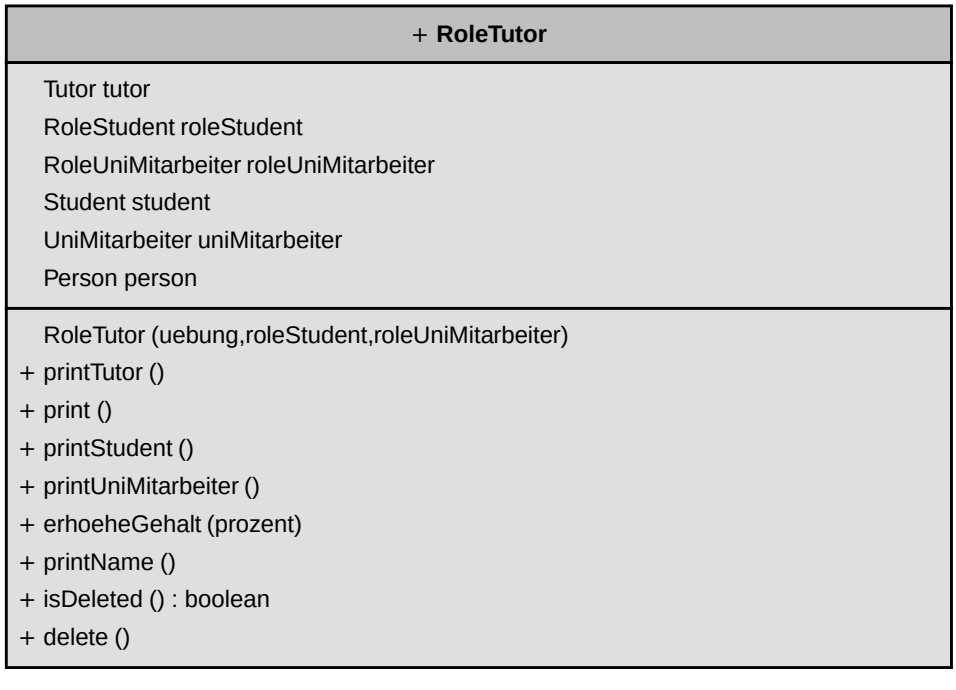


Abb. 94 Die Struktur der Klasse **RoleTutor**

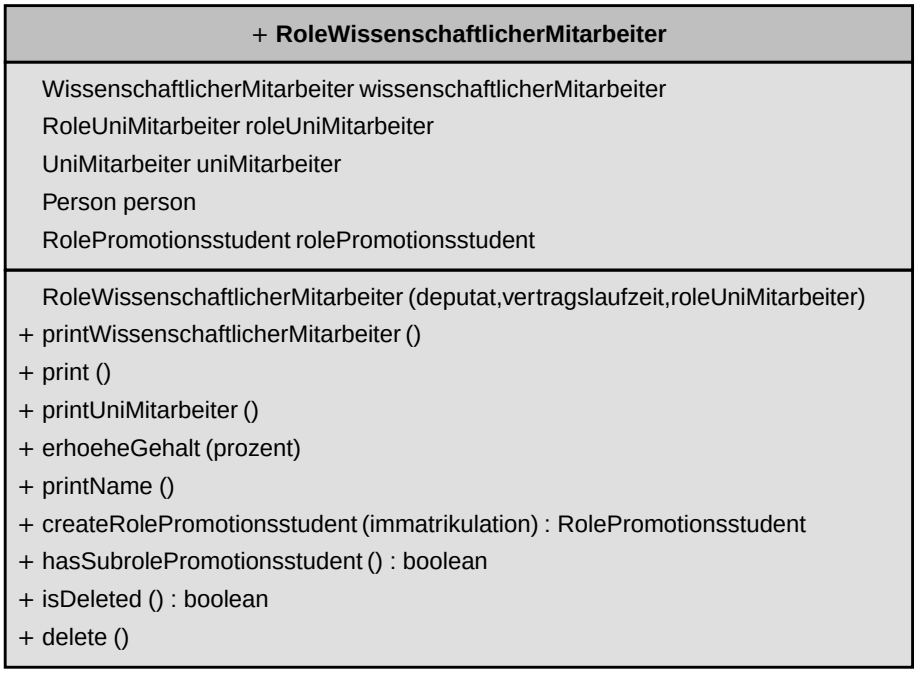


Abb. 95 Die Struktur der Klasse **RoleWissenschaftlicherMitarbeiter**

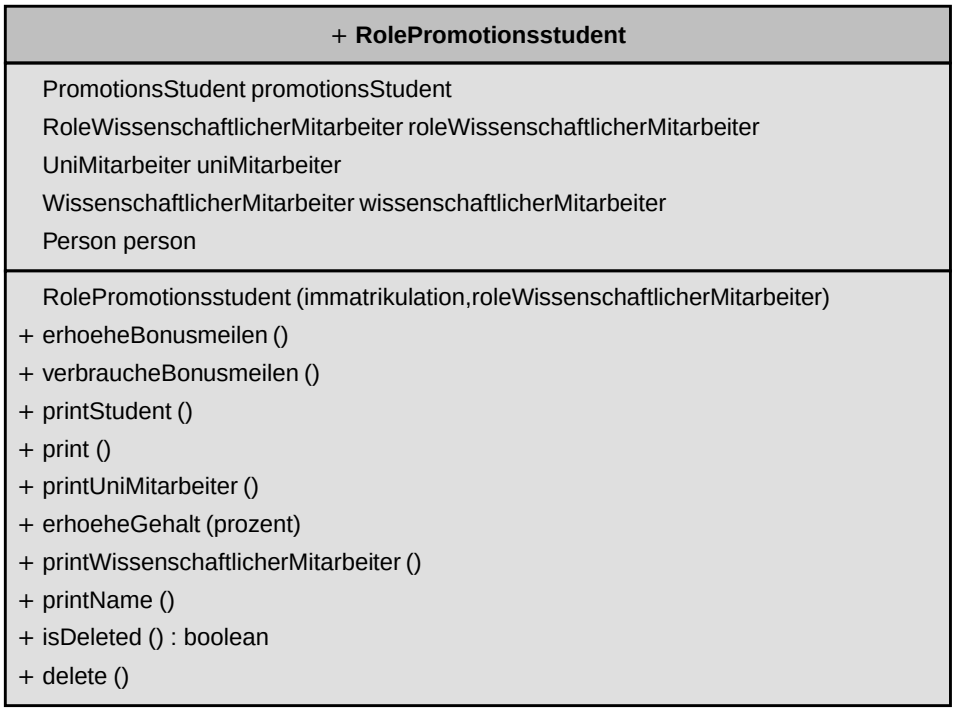


Abb. 96 Die Struktur der Klasse **RolePromotionsstudent**

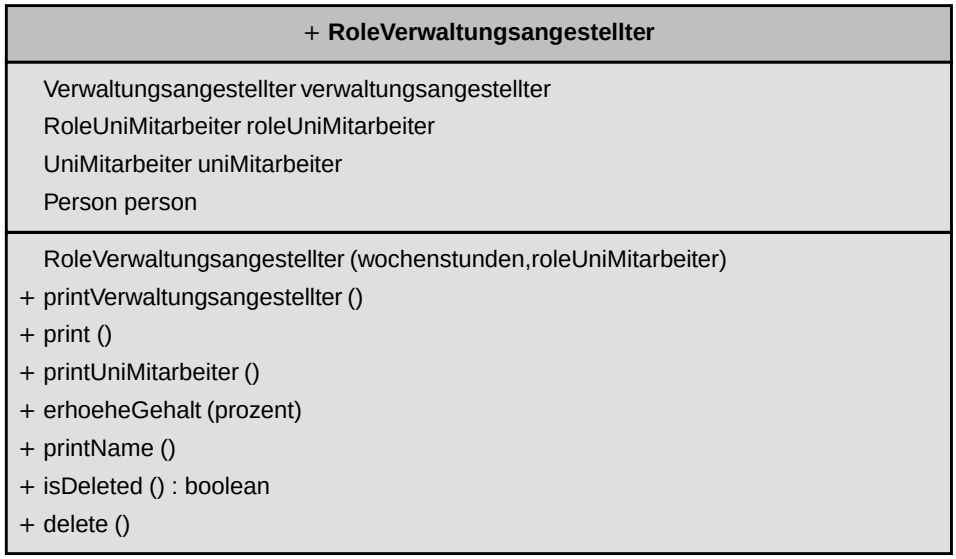


Abb. 97 Die Struktur der Klasse **RoleVerwaltungsangestellter**

Bei der Klasse **RolePromotionsstudent** läßt sich die Frage stellen, ob es aus fachlicher Sicht sinnvoll ist, für einen Promotionsstudenten das Gehalt zu erhöhen. Da zwischen Rollen eine *ist-ein*-Beziehung besteht, muß **RolePromotionsstudent** diese Operation anbieten. Durch eine Redefinition von `erhoeheGehalt` in **RolePromotionsstudent** könnte man aber z.B. dafür sorgen, daß die Operation wirkungslos bleibt. Selbstverständlich kann für die Person in ihrer Rolle als wissenschaftlicher Mitarbeiter aber weiterhin eine Gehaltserhöhung vorgenommen werden.

Die Klasse **Mitarbeiterverwaltung** in Abb. 98 (S. 189) ist eine Container-Klasse, die auf dem Interface **RoleUniMitarbeiterInt** basiert. Daher können alle Rollenklassen, die dieses Interface implementieren, in die Mitarbeiterverwaltung aufgenommen werden. Damit wird über die statische Typprüfung des JAVA Compilers sichergestellt, daß weder die abstrakten Rollen (hier **RolePerson** und **RoleUniMitarbeiter**) noch Studenten in der Mitarbeiterverwaltung enthalten sein können. Übernimmt dagegen ein Student eine Tutorenstelle, kann er in dieser Rolle in die Mitarbeiterverwaltung integriert werden.

Ein **Person**-Objekt kann in der Mitarbeiterverwaltung mit unterschiedlichen Rollen enthalten sein, was beispielsweise bei der Ausführung der Operation `erhoeheGehaltFuerAlle` bedeutet, daß auch für jede dieser Rollen eine Gehaltserhöhung durchgeführt wird. Hier ist im Rahmen der Analyse der Applikation, die das Rollenmodell benutzt, zu entscheiden, inwieweit dieses mit der Realität übereinstimmt. Als alternative Modellierung der Universitätsverwaltung könnte man einem **Tutor**-Objekt ein eigenes Gehaltsattribut zuordnen, auf das sich dann eine in **RoleTutor** redefinierte Operation `erhoeheGehalt` bezieht. Jetzt wird zwar nicht mehr die Funktionalität der `erhoeheGehalt`-Operation aus **RoleUniMitarbeiter** genutzt, es besteht aber weiterhin die Möglichkeit, über eine **RoleUniMitarbeiterInt**-Referenz für eine Tutorenrolle die `erhoeheGehalt`-Operation aufzurufen. Diesen Ansatz sollte man immer dann benutzen, wenn durch die Übernahme einer weiteren Rolle ein zusätzliches Gehalt entsteht.

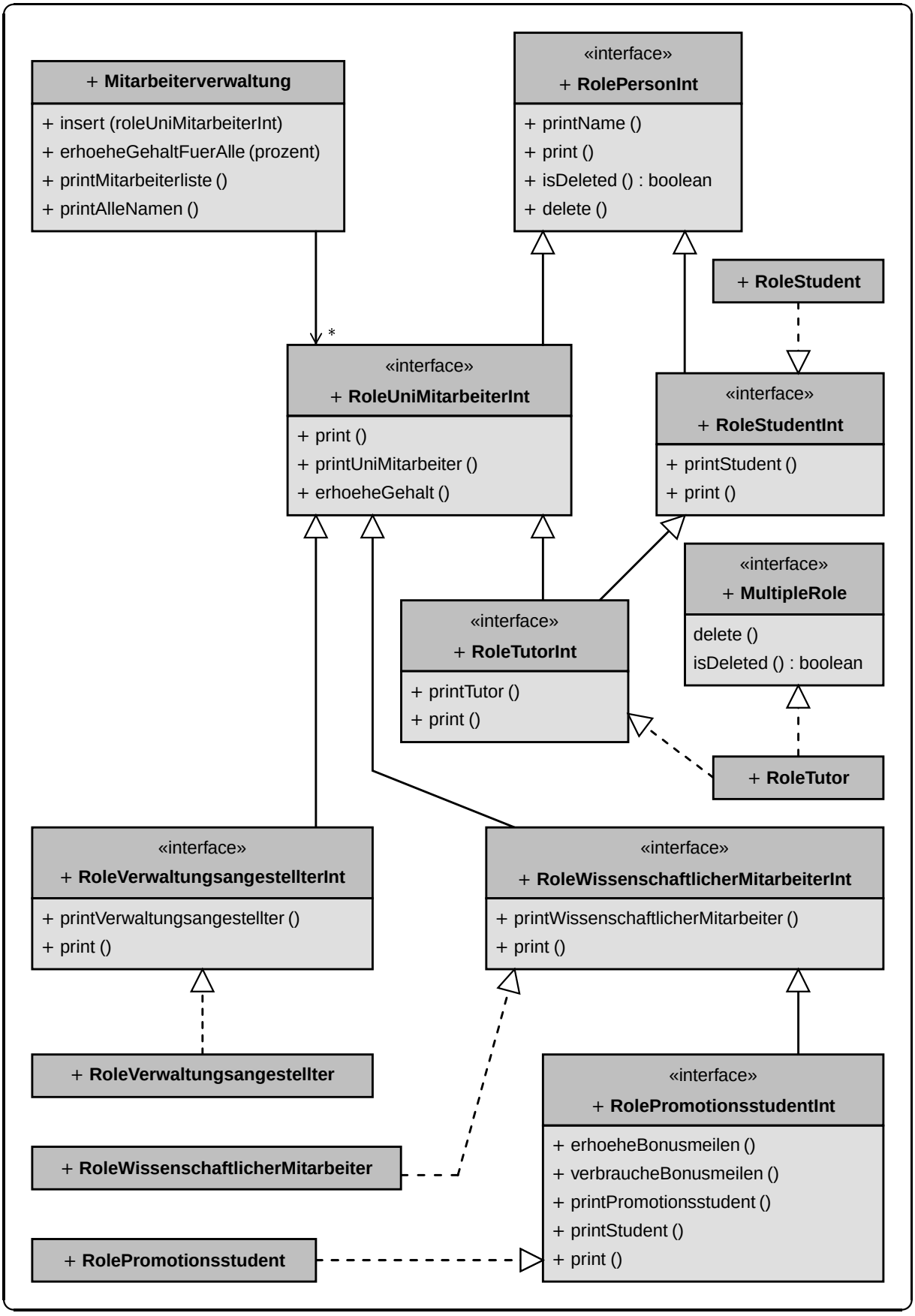


Abb. 98 Die Container-Klasse **Mitarbeiterverwaltung**

5.8 Weitere Ergänzungen der Funktionalität des Rollenmodells

5.8.1 Restriktionen

Die Festlegung, welche Restriktionen ein Fachkonzept einhalten soll, ist Aufgabe der Analysephase. Diese Restriktionen sind dann im Entwurf und der Implementierung entsprechend umzusetzen. Verwendet die Applikation ein Rollenmodell, so können die dort unterstützten Restriktionen im Rahmen des Generierungsprozesses direkt implementiert werden. Alle zusätzlich erforderlichen Restriktionen sind von den anderen Bestandteilen der Applikation sicherzustellen.

Neben den in Abschnitt 5.6 (S. 174) vorgestellten Restriktionen lassen sich für ein allgemeines Rollenmodell natürlich weitere Restriktionen definieren, die dann auf der Ebene der Rollenmodellbeschreibung spezifizierbar sind, und innerhalb des Generierungsprozesses umgesetzt werden. Ein Beispiel hierfür ist eine **excludes**-Restriktion, über die man festlegen kann, welche Rollen nicht mehr angenommen werden dürfen, wenn eine bestimmte andere Rolle bereits existiert.

Für das Rollenmodell einer Universitätsverwaltung aus Abb. 88 (S. 179) sind die folgenden **excludes**-Restriktionen denkbar:

- **Student excludes WissenschaftlicherMitarbeiter**
  Ein Student darf nicht wissenschaftlicher Mitarbeiter werden. Dies ist innerhalb der Universitätsverwaltung bisher möglich, wenn ein Student eine Tutorenrolle übernimmt. Hierfür benötigt er ein **RoleUniMitarbeiter**-Objekt, über das er dann die Rolle eines wissenschaftlichen Mitarbeiters erzeugen könnte.
- **Student excludes Verwaltungsangestellter**
  Ein Student darf nicht Verwaltungsangestellter werden.
- **UniMitarbeiter excludes Student**
  Ein Universitätsmitarbeiter darf sich nicht als Student immatrikulieren. Die umgekehrte **excludes**-Restriktion darf jedoch nicht gelten, da andernfalls ein Student keine Tutorentätigkeit übernehmen könnte.

Diese drei **excludes**-Restriktionen enthalten implizit weitere **excludes**-Restriktionen für Unterrollen. Aus der ersten **excludes**-Restriktion folgt beispielsweise, daß ein Student nicht die Rolle eines Promotionsstudenten übernehmen kann, weil er hierfür die Rolle eines wissenschaftlichen Mitarbeiters benötigt.

5.8.2 Verwaltungsoperationen

Eine Applikation erhält für jede neu erzeugte Rolle eine entsprechende Referenz. Daher besitzt sie grundsätzlich alle Referenzen, um auf die Unterrollen der Wurzelrollenobjekte zuzugreifen. Trotzdem kann es natürlich praktisch sein, sich ausgehend von einer Rollenreferenz weitere Rollenreferenzen besorgen zu können. Innerhalb des Rollenmodells bieten sich hierfür die folgenden Verwaltungsoperationen an:

- `RoleY getRoleY()` für alle Oberrollen **RoleY** einer Rollenklasse **RoleX**.
  Innerhalb des Rollenmodells aus Abb. 88 (S. 179) wären dies für die Klasse **RolePromotionsstudent** die folgenden drei Operationen:
  - `getRoleWissenschaftlicherMitarbeiter`
  - `getRoleUniMitarbeiter`
  - `getRolePerson`
- `RoleZ getRoleZ()` für alle direkten Unterrollen `RoleZ` einer Rollenklasse **RoleX** mit der Kardinalität 0..1.
  Für die Rolle **RoleUniMitarbeiter** sind dies drei Operationen:
  - `getRoleTutor`
  - `getRoleWissenschaftlicherMitarbeiter`
  - `getRoleVerwaltungsangestellter`
- `RoleZ[] getRoleZ()` für alle direkten Unterrollen einer Rollenklasse **RoleX** mit einer Kardinalität größer als 1.
  Für die Klasse **RoleStudent** ist dies die Operation `getRoleTutor`.

5.8.3 Die Mehrfachverwendung einer korrespondierenden Klasse in einem Rollenmodell

Eine korrespondierende Klasse kann auf direktem Weg nicht mehrfach in einem Rollenmodell verwendet werden, da durch den Generierungsprozeß eine feste Namensbindung zwischen dieser und der erzeugten Rollenklasse hergestellt wird (aus der korrespondierenden Klasse **X** wird die Rollenklasse **RoleX** erzeugt).

Betrachtet man ein Rollenmodell, in dem Kunden- und Verkäuferrollen auftreten, so ist es sinnvoll, daß unterschiedliche Oberrollen beispielsweise eine Kundenrolle übernehmen können. Liegen die Oberrollen **Firma** und **Person** vor, so können beide die Kundenrolle ausüben.

Eine Lösung dieses Problems wäre, in der Grammatik zur Beschreibung von Rollenmodellen die Spezifikation eines Präfixes für einen Rollennamen zu erlauben:

```
$root Person hasroles$ Privat::Kunde root$
$root Firma  hasroles$ Firmen::Kunde root$
```

Alternativ kann das Problem aber auch unter Verwendung der vorhandenen Modellierungskonzepte gelöst werden. In Abb. 99 (S. 192) ist ein erster Ansatz beschrieben. Es wird eine abstrakte Rolle Kunde bereitgestellt, die die gesamte Funktionalität einer späteren Kundenrolle enthält. Diese abstrakte Rolle kann entweder eine Firmenkundenrolle oder eine Privatkundenrolle übernehmen. Die zugehörigen korrespondierenden Klassen besitzen keine weitere Funktionalität, ihre Aufgabe liegt hier nur in der Bereitstellung neuer Namen für Kundenrollen. Eine Firma kann nun eine Firmenkundenrolle annehmen und eine Person eine Privatkundenrolle. Durch die **exor**-Restriktion wird sichergestellt, daß der abstrakten Rolle **Kunde** höchstens eine der beiden Unterrollen zuzuordnen ist.

Eine weitere Alternative besteht darin, auf die abstrakte Rolle **Kunde** ganz zu verzichten, und stattdessen die korrespondierenden Klassen **Firmenkunde** und **Privatkunde** jeweils die abstrakte Klasse **Kunde** erweitern zu lassen.

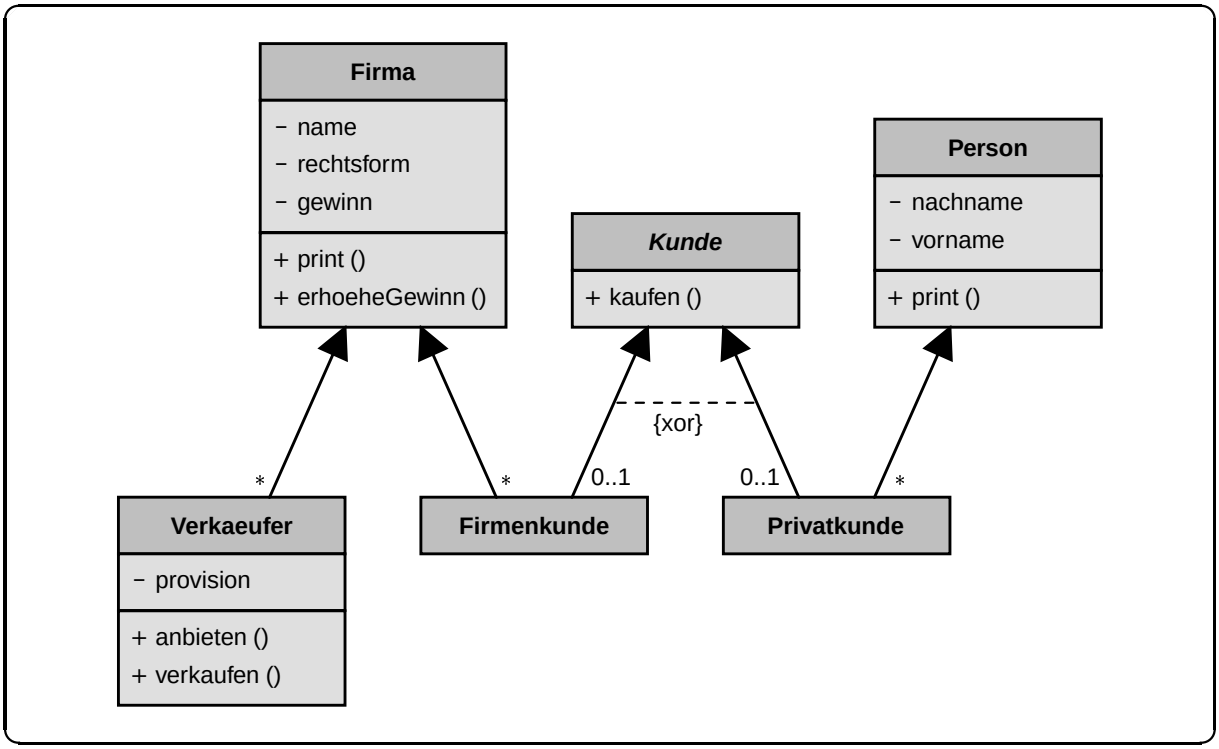


Abb. 99 Die Mehrfachverwendung einer Kundenrolle

5.8.4 Die Verwendung korrespondierender Klassen in mehreren Rollenmodellen

Befinden sich die Rollenmodelle in demselben Paket, so lassen sich mit den Ansätzen aus Abschnitt 5.8.3 (S. 191) korrespondierende Klassen in unterschiedlichen Rollenmodellen anwenden. Soll dagegen jedes Rollenmodell einem eigenen Paket zugeordnet werden, sind zwei Realisierungsansätze denkbar:

- Die korrespondierenden Klassen werden als **`public`** deklariert.
  Damit kann aus beliebigen anderen Paketen auf diese Klassen zugegriffen werden. Die Grammatik zur Generierung des Rollenmodells wird um zwei Paketangaben erweitert, das Quellpaket und das Zielpaket. Das Quellpaket legt fest, in welchem Paket sich die korrespondierenden Klassen befinden, das Zielpaket definiert die Paketzuordnung der erzeugten Rollenklassen. Zusätzlich könnte man noch die Spezifikation eines vollqualifizierten Klassennamens[20] bei der Angabe einer korrespondierenden Klasse erlauben. Liegt jetzt ein einfacher Klassenname vor, so wird das Quellpaket verwendet, um die korrespondierende Klasse zu finden, andernfalls der Paketname aus dem vollqualifizierten Klassennamen.
- Die korrespondierenden Klassen werden in das Zielpaket kopiert.
  Auch bei diesem Ansatz wird die Grammatik um die Angabe eines Quell- und eines Zielpakets sowie von vollqualifizierten Klassennamen für korrespondierende Klassen erweitert. Allerdings bleibt die Sichtbarkeit der korrespondierenden Klassen auf das eigene Paket beschränkt, was zur Folge hat, daß Rollenklassen und korrespondierende Klassen in dem-

[20] Ein vollqualifizierter Klassenname enthält neben dem einfachen Klassennamen noch den Paketnamen.

selben Paket enthalten sein müssen. Bei der Generierung des Rollenmodells werden die korrespondierenden Klassen in das Zielpaket kopiert und ihre **`package`**-Anweisungen entsprechend geändert. Nach der Erzeugung der JAVA `class`-Dateien sollten die kopierten Quelltexte der korrespondierenden Klassen wieder gelöscht werden, damit nicht mehrere Quelltextversionen einer korrespondierenden Klasse vorliegen.

Der erste Realisierungsansatz hat den Nachteil, daß die korrespondierenden Klassen von beliebigen Klassen aus jedem Paket verwendbar sind. Beim zweiten Ansatz liegen für eine Quelltextdatei mehrere JAVA `class`-Dateien vor. Wird also der Quelltext geändert, muß sichergestellt werden, daß auch alle zugehörigen JAVA `class`-Dateien neu erzeugt werden. Falls hierfür eine entsprechende Unterstützung durch eine Entwicklungsumgebung vorhanden ist, ist dieser Ansatz zu bevorzugen, da die korrespondierenden Klassen nur für ihre Rollenklassen sichtbar sein sollten.

5.8.5 Die vollständige Kapselung korrespondierender Klassen

Eine korrespondierende Klasse **X** wird als vollständig gekapselt bezeichnet, wenn Zugriffe auf Objekte dieser Klasse ausschließlich den folgenden Objekten erlaubt sind:

- Das zugehörige Rollenobjekt der Rollenklasse **RoleX**.
- Objekte einer korrespondierenden Klasse **Y**, wobei **Y** eine Assoziation zu **X** entsprechend den Regeln aus Abschnitt 5.2.5 (S. 119) besitzt.

Bei dem verwendeten Generierungsprozeß ist es erlaubt, daß eine öffentliche Methode einer korrespondierenden Klasse **X** auch den Ergebnistyp **X** besitzt. Damit kann eine Referenz eines korrespondierenden Objekts durch die zugehörige Methode der Rollenklasse **RoleX** an die aufrufende Applikation zurückgeliefert werden. Unter der Annahme, daß die Applikation in einem eigenen Paket liegt, gibt es zwei Varianten, um eine vollständige Kapselung der korrespondierenden Klassen zu erreichen.

- Die korrespondierenden Klassen sind nur paketintern sichtbar (vgl. den zweiten Realisierungsansatz in Abschnitt 5.8.4 (S. 192)). In diesem Fall liefert die Übersetzung der Applikation einen Übersetzungsfehler, falls eine Referenz einer korrespondierenden Klasse in einer Applikationsklasse verwendet werden soll.
- Sind die korrespondierenden Klassen dagegen als **`public`** deklariert, kann man für die korrespondierenden Klassen eines Rollenmodells eine weitere Restriktion definieren, die dann vom Rollenmodellgenerator zu überprüfen ist: öffentliche Methoden einer korrespondierenden Klasse dürfen weder als Parameter- noch als Ergebnistyp eine korrespondierende Klasse aus dem zugehörigen Rollenmodell besitzen.

Die erste Variante ist einfacher zu realisieren, während die zweite Vaiante bereits auf semantischer Ebene überwacht, daß keine unerlaubten Beziehungen zu korrespondierenden Objekten aufgebaut werden.

Die vollständige Kapselung korrespondierender Klassen ist eine wesentliche Voraussetzung, um einen Objekttausch zur Laufzeit zu unterstützen (vgl. Abschnitt 5.8.7 (S. 194)).

5.8.6 Die Umsetzung von Objektkollaborationen

Ein Rollenobjekt `roleX` kann nur dann mit einem Objekt `roleY` der Rollenklasse **RoleY** zusammenarbeiten (d.h. Methoden von `roleY` aufrufen), wenn es eine Referenz auf `roleY` besitzt. Durch diese Referenz ist aber nicht nur der Zugriff auf das Rollenobjekt, sondern auch auf die Operationen aller Oberrollenobjekte von `roleY` möglich. Will man den Zugriff auf die öffentlichen Methoden der zu **RoleY** korrespondierenden Klasse **Y** beschränken, so ist die Definition einer entsprechenden Schnittstelle sinnvoll.[21] Die bereits erzeugte Schnittstelle **RoleYInt** ist für diesen Zweck nicht geeignet, da sie wie eine **RoleY**-Referenz den Zugriff auf alle Oberrollenmethoden erlaubt. Daher sollte zusätzlich für jede korrespondierende Klasse **Z** eine Schnittstelle **RoleZCol** generiert werden, die nur die öffentlichen Methoden von **Z** enthält, und selbst keine anderen Schnittstellen erweitert. Die Rollenklasse **RoleZ** implementiert dann neben **RoleZInt** auch **RoleZCol**. Damit kann eine Methode statt eines **RoleZ**-Parameters einen **RoleZCol**-Parameter definieren. Diesem Parameter kann jetzt eine **RoleZ**-Referenz zugewiesen werden, der Parametertyp verhindert aber einen Zugriff auf die Oberrollenmethoden des **RoleZ**-Objekts.

5.8.7 Der Objekttausch zur Laufzeit

Für eine JAVA-Applikation ist es in der Regel zur Laufzeit nicht möglich, ein Objekt einer Klasse durch eine neue Objektversion zu ersetzen, um beispielsweise Implementierungsfehler zu beheben, oder eine effizientere Implementierung nutzen zu können. Das Hauptproblem liegt in der Eigenschaft begründet, daß für ein Objekt beliebig viele Objektreferenzen existieren können. Um das Objekt auszutauschen, müßten auch alle Referenzen ersetzt werden. Es gibt in JAVA aber keine Operation, um gezielt auf alle Referenzen eines Objekts zuzugreifen.

Das für die Umsetzung des Rollenmodells gewählte Entwurfskonzept erlaubt einen vergleichsweise einfachen Austausch der Rollenfunktionalität. Zustand und Funktionalität einer Rolle sind über die korrespondierende Klasse beschrieben. Zur Laufzeit ist jedem Rollenobjekt genau ein korrespondierendes Objekt zugeordnet. Ein Austauch der Rollenfunktionalität läßt sich damit auf die Ersetzung des korrespondierenden Objekts zurückführen. Unter der Annahme einer vollständigen Kapselung der korrespondierenden Klassen sind zwei Fälle zu unterscheiden:

- Zwischen den korrespondierenden Klassen gibt es innerhalb des Rollenmodells keine Assoziationen.
  Damit existiert für jedes korrespondierende Objekt genau eine Referenz und zwar innerhalb des zugehörigen Rollenobjekts.
- Zwischen den korrespondierenden Objekten liegen Assoziationen vor.
  In diesem Fall sind mehrere Referenzen auf ein korrespondierendes Objekt vorhanden.

Um alle korrespondierenden Objekte einer Rollenklasse auszutauschen, ist es notwendig, auf alle Rollenobjekte zugreifen zu können. Hierfür bietet es sich an, für eine Rollenklasse eine eigene Objektverwaltung in Form einer Liste zu generieren. Jede Rollenobjekterzeugung führt zu einem Eintrag der Referenz in dieser Liste. Bei einem Objekttausch muß der Zustand des alten Objekts in das neue Objekt transferiert werden. Daher benötigt eine korrespondierende Klasse

[21] Um eine vollständige Kapselung der korrespondierenden Klasse **Y** zu erreichen, darf eine Referenz dieser Klasse nicht verwendet werden.

die zusätzlichen Methoden `exportState` und `importState`. Geht man davon aus, daß die neue Version der korrespondierenden Klasse nicht als Unterklasse der alten Version definiert wurde, dann kann eine Referenz der neuen Version nicht einfach einer Referenz der alten Version zugewiesen werden. Am einfachsten ist dieses Problem zu lösen, indem man bei der Generierung einer Rollenklasse **RoleX** nicht mehr eine Referenz des Typs **X** erzeugt, sondern stattdessen den Interface-Typ **RoleXCol** verwendet. Die verschiedenen Versionen einer korrespondierenden Klasse sind dann nur um eine **`implements`** `RoleXCol`-Anweisung zu ergänzen. Jede Rollenklasse erhält die neue Klassenoperation `exchangeObjects(String className)` mit der folgenden Struktur:

- Laden der Klasse `className`; `className` soll dabei eine neue Version der korrespondierenden Klasse **X** darstellen.
- Durchlaufen der Liste der Rollenobjekte:
  - Erzeugung eines neuen korrespondierenden Objekts.
  - Export des Zustands aus dem alten korrespondierenden Objekt.
  - Import der Zustandsinformation in das neue korrespondierende Objekt.
  - Zuweisung der Referenz des neuen korrespondierenden Objekts an das `RoleXCol`-Attribut.

Falls zwischen korrespondierenden Klassen Assoziationen vorliegen, besteht das Problem, für ein korrespondierendes Objekt `x` die in anderen korrespondierenden Objekten y_i vorhandenen Referenzen zu finden. Dieses Problem läßt sich vergleichsweise einfach umgehen, indem die y_i-Objekte nicht direkt auf `x` zugreifen, sondern den ***Umweg*** über das zugehörige Rollenobjekt nehmen. Durch diese Form der indirekten Referenzierung gibt es nur noch genau eine direkte Referenz auf `x`.

5.8.8 Die Migration von Rollenobjekten

Zur Realisierung der Rollenobjektmigration ist in einer Rollenklasse **RoleX** eine neue Verwaltungsoperation

`boolean` `migrateRoleX (RoleY`$_1$ `ry`$_1$`,...,RoleY`$_n$ `ry`$_n$`)`

notwendig. Die Parameter ry_1 bis ry_n sind die Referenzen auf die neuen Oberrollenobjekte. Die `migrateRoleX`-Methode hat die folgende Struktur:

- Für alle neuen Oberrollenobjekte wird getestet, ob sie das **RoleX**-Objekt übernehmen können. Hier sind genau diejenigen Tests auszuführen, die in den entsprechenden `createRoleX`-Operationen enthalten sind.
- Alle Tests waren erfolgreich:
  - Das **RoleX**-Objekt wird als Unterrollenobjekt bei den neuen **RoleY**$_i$-Objekten installiert.
  - Die Referenzen auf das migrierte **RoleX**-Objekt werden in den alten **RoleY**$_i$-Objekten auf `null` gesetzt.
  - Der Ergebniswert `true` wird zurückgeliefert.
- Mindestens ein Test ist fehlgeschlagen:
  - Eine Rollenobjektmigration ist nicht möglich; der Ergebniswert der Operation ist `false`.

5.9 Die Integration des Rollenmodells in den Software-Entwicklungsprozeß

5.9.1 Die Analysephase

Innerhalb der Analysephase werden zwei Ziele verfolgt:

- Die Identifikation der Rollenmodelle des Analysemodells.
- Die Beschreibung der Objektkollaborationen innerhalb eines Rollenmodells.

Das erste Ziel gehört zum statischen Analysemodell, das zweite zum dynamischen Analysemodell. Ein Rollenmodell zeichnet sich dadurch aus, daß die zugehörigen Rollen keine Beziehungen zu Rollen aus anderen Rollenmodellen besitzen. Aus diesem Grund wird es in der Regel erforderlich sein, statisches und dynamisches Analysemodell in mehreren Iterationszyklen zu erstellen. Zunächst wird mit der Ermittlung der Rollenmodelle begonnen. Für die einzelnen Rollen sind genau diejenigen Operationen zu identifizieren, die von späteren Anwendern benötigt werden, d.h. das nach außen sichtbare Applikationsverhalten. Liegt das erste statische Modell vor, lassen sich die Rollenkollaborationen analysieren. Hierbei kann sich herausstellen, daß bestimmte Rollenmodelle noch zusammenzufassen sind, wodurch ein neues statisches Modell entsteht. Jedes resultierende Rollenmodell wird einem eigenen Paket zugeordnet.

5.9.2 Die Entwurfsphase

Bei der Verwendung einer Drei-Schichten-Architektur sollten die Klassen der einzelnen Schichten in disjunkten Paketen liegen, wobei natürlich pro Schicht mehrere Pakete verwendet werden dürfen. Die in der Analysephase spezifizierten Rollenmodelle bilden die Grundlage für die Applikationsschicht. In der Entwurfsphase sind die folgenden Ergänzungen an den Rollenmodellen vorzunehmen:

- Es müssen zusätzliche, öffentliche Operationen spezifiziert werden, die für das Zusammenspiel mit der Präsentationsschicht erforderlich sind. In Bezug auf das MVC-Muster (vgl. Abschnitt 2.4 (S. 12)) stellen die Rollenklassen die **Model**-Funktionalität dar. Die **Controller**- und **View**-Klassen liegen in der Präsentationsschicht und benötigen den Zugriff auf die Rollenklassen.
- Die Rollenklassen müssen (möglicherweise) um interne Zugriffsoperationen auf die Klassen der Datenzugriffsschicht ergänzt werden.
- Alle weiteren internen Operationen der Rollenklassen sind zu spezifizieren.

Parallel zum statischen Modell der Entwurfsschicht ist das dynamische Analysemodell in das dynamische Entwurfsmodell zu überführen und zu verfeinern.

5.9.3 Die Implementierungsphase

Für die Applikationsschicht sind innerhalb der Implementierungsphase die korrespondierenden Klassen zu implementieren. Der zweite Schritt besteht in der Erstellung der Beschreibungsdateien für die Rollenmodelle aus der Analysephase. Danach können die Rollenklassen generiert und übersetzt werden.

5.10 Die Software-Architektur zur Generierung von Rollenmodellen

5.10.1 Die Gesamtstruktur

Das statische Entwurfsmodell des Rollenmodellgenerators ist in Abb. 101 (S. 198) dargestellt, wobei allerdings einige Hilfsklassen und auch die Exception-Klassen zur Verbesserung der Übersichtlichkeit weggelassen wurden. Auch wurde bei der Klasse **RoleModelSingleInheritance** auf eine Angabe ihrer Assoziationen zu anderen Klassen verzichtet, da in der Folge die Funktionalität der allgemeineren Klasse **RoleModelMultipleInheritance** betrachtet wird. In **RoleModelSingleInheritance** kann bei der Generierung eines Rollenmodells auf verschiedene Tests verzichtet werden, die bei der Einfachvererbung im Vergleich zur Mehrfachvererbung nicht notwendig sind. Prinzipiell besitzt **RoleModelSingleInheritance** Assoziationen zu denselben Klassen wie **RoleModelMultipleInheritance**. Die Implementierung des Entwurfsmodells umfaßt ca. 7100 Quelltextzeilen (*Lines of Code*).

Die Klasse **RoleModelUI** (vgl. Abb. 100) stellt den Ausgangspunkt für den Generierungsprozeß dar. Bei dem Aufruf ihrer `main`-Methode wird als Parameter der Name der Datei übergeben, die die Beschreibung des Rollenmodells enthält.

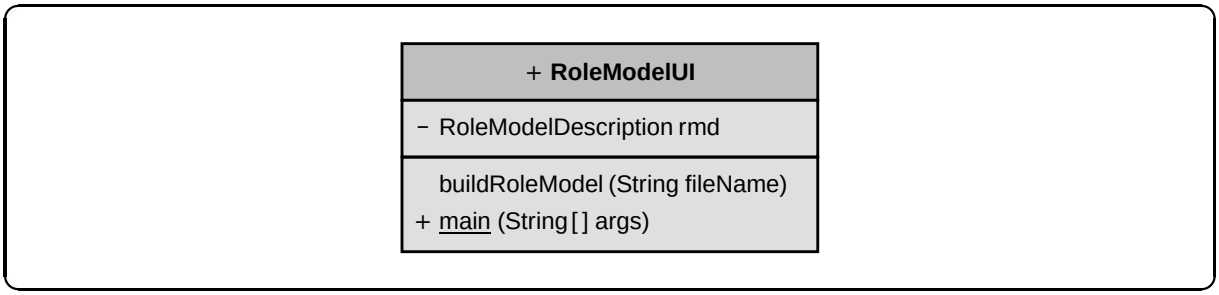


Abb. 100 Die Klasse **RoleModelUI**

Nach der Erzeugung eines **RoleModelUI**-Objekts und dem Aufruf der `buildRoleModel`-Methode läuft der Generierungsprozeß in den folgenden Schritten ab:

- Ein **RoleModelParser**-Objekt wird erzeugt. Der zugehörige Konstruktor erstellt ein **TokenStream**-Objekt, das seinerseits ein **Scanner**-Objekt anlegt. Dieses zerlegt die Rollenmodellbeschreibung in die Eingabesymbole (*Token*) und liefert an das **TokenStream**-Objekt ein **Token**-Array zurück.
- Im nächsten Schritt wird die Syntaxprüfung durchgeführt. Falls sie erfolgreich war, stellt sie als Ergebnis ein **RoleModelDescription**-Objekt zur Verfügung.
- Ein **RoleModelDescription**-Objekt enthält alle für den Generierungsprozeß relevanten Informationen. Die einzelnen Rollen sind über **RoleDescription**-Objekte beschrieben. Auf die genaue Struktur eines **RoleModelDescription**-Objekts wird in Abschnitt 5.10.4 (S. 203) eingegangen. Ausgehend von der Rollenmodellbeschreibung wird ein **InheritanceStructure**-Objekt aufgebaut.
- Ein **InheritanceStructure**-Objekt enthält intern eine Adjazenzmatrix des Graphen, der über das Rollenmodell definiert wird. Zusätzlich existiert eine Pfadmatrix, die jeweils die Anzahl

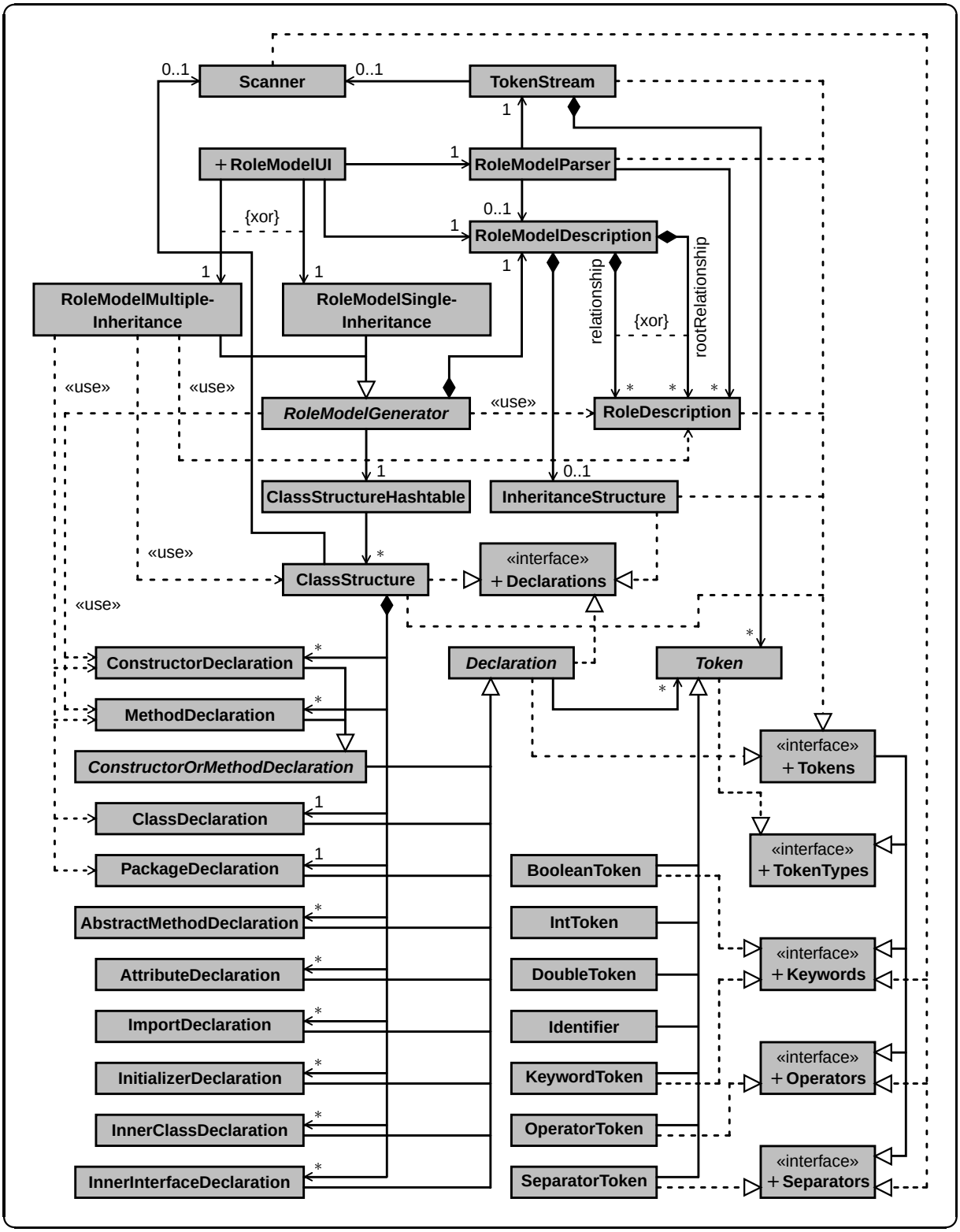


Abb. 101 Das Entwurfsmodell des Rollenmodellgenerators

der Pfade zwischen zwei Knoten des Graphen abspeichert. Beim Aufbau der Pfadmatrix wird getestet, ob der Graph zyklenfrei ist. Falls dieses nicht der Fall ist, liegt ein inkonsistentes Rollenmodell vor.

- Im nächsten Schritt wird überprüft, ob die **nocreation**-Restriktionen konsistent sind (vgl. Abschnitt 5.6.3 (S. 179)).
- Im Erfolgsfall wird die Rollenmodellbeschreibung dahingehend untersucht, ob ein Modell mit Einfach- oder Mehrfachvererbung vorliegt. Für ein Modell mit Mehrfachvererbung[22] werden zwei weitere Tests vorgenommen: falls der Graph transitive Kanten enthält (vgl. S. 151) oder aber fehlerhafte **exor**-Restriktionen definiert wurden (vgl. Abschnitt 5.6.2 (S. 174)), wird die Rollenmodellerzeugung abgebrochen.
- Andernfalls erfolgt die Erzeugung eines **RoleModelMultipleInheritance**-Objekts. Innerhalb des Oberklassenkonstruktors[23] wird über den Aufruf der Methode `constructHashtable` die interne Hash-Tabelle aufgebaut, die die **ClassStructure**-Objekte für die korrespondierenden Klassen aufnimmt.
- Bei der Erzeugung eines **ClassStructure**-Objekts findet eine Zerlegung des Quelltextes der Klasse in ihre deklarierten Bestandteile statt. Für den weiteren Generierungsprozeß spielen hierbei die Klassen-, die Paket-, die Konstruktor- und die Methodendeklarationen eine zentrale Rolle. Für jedes in der Klasse deklarierte Element besitzt ein **ClassStructure**-Objekt entsprechende **Declaration**-Objekte.
- Nach einem Test, ob alle im Rollenmodell redefinierten Methoden gültig sind (vgl. S. 154), findet der Aufruf der `generate`-Methode statt. Diese generiert unter Verwendung der **ClassStructure**-Objekte zunächst alle Wurzelrollen und danach alle Unterrollen des Rollenmodells.

In den folgenden Abschnitten wird kurz auf die Struktur der Schnittstellen und Klassen aus Abb. 101 (S. 198) eingegangen. Zusätzlich findet eine Beschreibung der wichtigsten Methoden der einzelnen Klassen statt.

5.10.2 Die Schnittstellendefinitionen

Da JAVA[24] keine Aufzählungstypen kennt, wurden die in Abb. 102 (S. 200) dargestellten Schnittstellen definiert. **Keywords** enthält neben den Schlüsselwörtern aus der Rollenmodellbeschreibung Konstantendefinitionen für alle JAVA-Schlüsselwörter. Für die JAVA-Operatoren und JAVA-Separatoren besitzen die Schnittstellen **Operators** sowie **Separators** entsprechende Konstanten. Die Schnittstelle **TokenTypes** stellt Konstanten für eine Klassifikation der Tokens zur Verfügung. Das Attribut `type` der abstrakten Klasse **Token** (vgl. Abb. 104 (S. 202)) wird mit einem Wert aus **TokenTypes** initialisiert. Die Schnittstelle **Declarations** definiert Konstanten für die verschiedenen Deklarationen einer Klasse. Das Attribut der abstrakten Klasse **Declaration** (vgl. Abb. 111 (S. 210)) besitzt einen Wert aus dieser Menge. Die Schnittstelle **Declarations** enthält keine Konstantendefinitionen für die Paket-, Import- und Klassendeklaration, da die zuzuordnenden Konstanten (package$, import$ und class$) bereits in **Keywords** existieren.

[22] Wie oben bereits erwähnt, wird auf die Einfachvererbung hier nicht weiter eingegangen.
[23] Dies ist der Konstruktor der abstrakten Klasse **RoleModelGenerator**.
[24] Zur Entwicklung der Rollenmodellgenerator-Software wurde die JAVA-Version 1.3.1 verwendet.

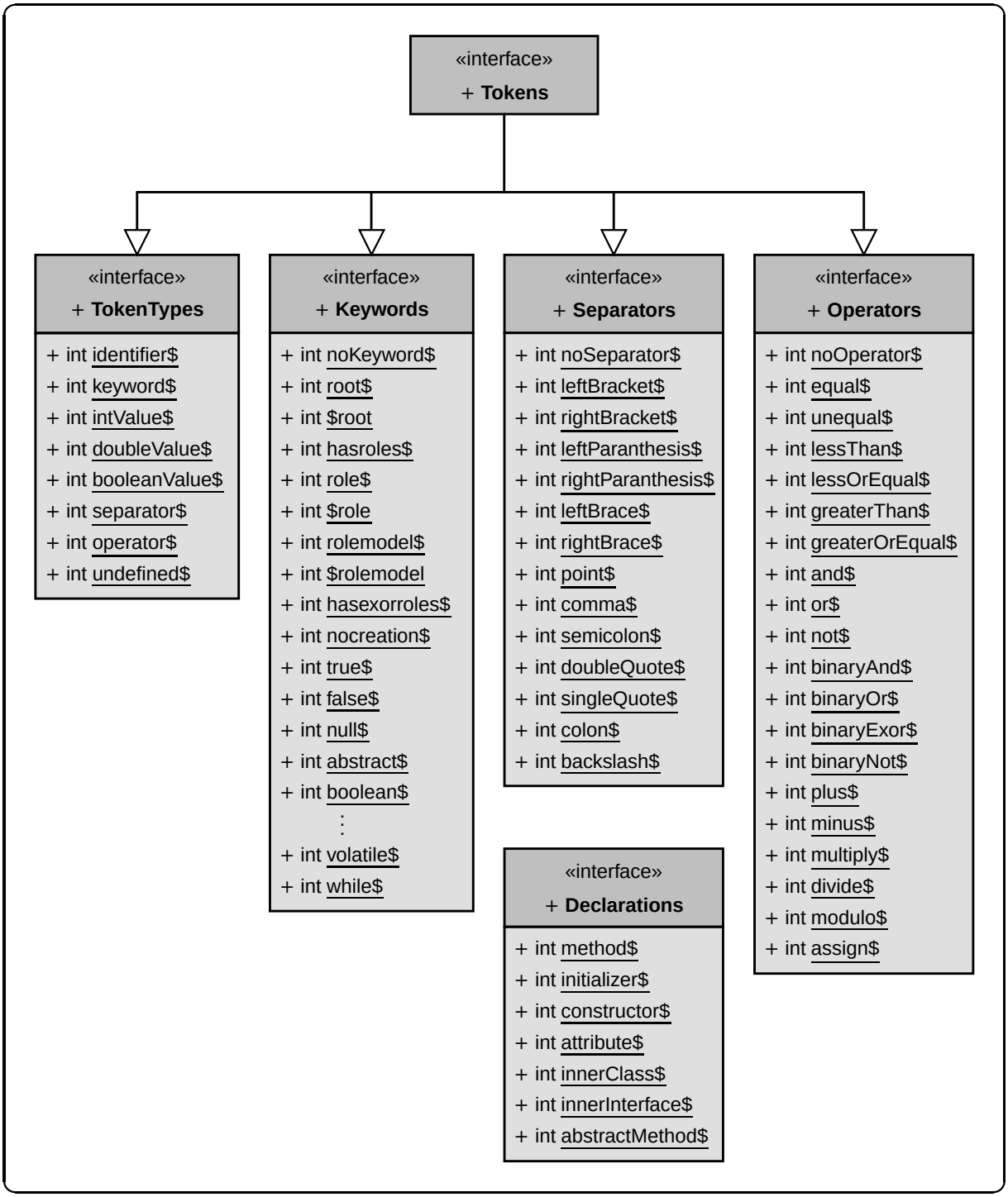


Abb. 102 Die Schnittstellendefinitionen

5.10.3 Die Klassen Scanner, TokenStream und Token

Die Klasse **Scanner** (vgl. Abb. 103) kann sowohl zur lexikalischen Analyse einer Rollenmodellbeschreibung als auch einer korrespondierenden Klasse eingesetzt werden. Bei der Erzeugung eines **Scanner**-Objekts wird der Name der Eingabedatei übergeben, die die Rollenmodelldefinition oder die korrespondierende Klasse enthält. Die `scan`-Methode liest die Datei ein und identifiziert die einzelnen Symbole. Für jedes Symbol wird ein Objekt des **Token**-Typs erzeugt. Die abstrakte Klasse **Token** und ihre Unterklassen sind in Abb. 104 (S. 202) dargestellt. Jedes **Token**-Objekt besitzt als Attribute die Zeichenkette, die zur Identifikation des Symbols diente, und den Symboltyp (eine **TokenTypes**-Konstante). Falls ein Schlüsselwort-, Separator- oder Operatorsymbol vorliegt, wird zusätzlich im `subtype`-Attribut die entsprechende **Keywords**-, **Separators**- oder **Operators**-Konstante abgelegt. Wenn die lexikalische Analyse erfolgreich war, liefert die `scan`-Methode als Ergebnis ein **Token**-Array, andernfalls wird eine **SyntaxError**-Exception erzeugt.

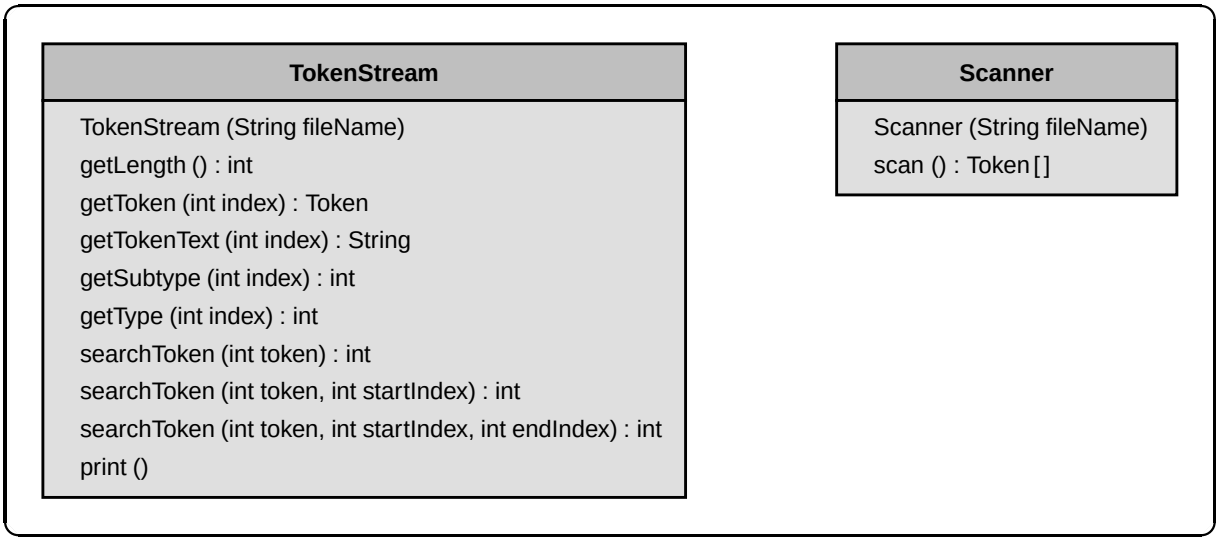


Abb. 103 Die Klassen **TokenStream** und **Scanner**

Die Unterklassen von **Token** (vgl. Abb. 104 (S. 202)) definieren **Token**-spezifische Attribute (wie z.B. die konkreten Werte `value` in den Klassen **IntToken**, **DoubleToken** und **BooleanToken**) sowie **Token**-spezifische Methoden. Ein wichtiges Beispiel ist die `isRoleIdentifier`-Methode der Klasse **Identifier**. Sie überprüft, ob es sich bei dem Bezeichner um einen gültigen Rollennamen handelt. Als Grundlage hierfür dient die *`RoleIdentifier`*-Regel der *`RoleModelDescription`*-Grammatik aus Abschnitt 5.7 (S. 181).

Ein Objekt der Klasse **TokenStream** (vgl. Abb. 103) enthält intern ein **Token**-Array. Neben den Zugriffsmethoden auf die Array-Elemente sind Suchmethoden realisiert, die als Ergebnis den Index des gesuchten **Token**-Objekts zurückliefern.

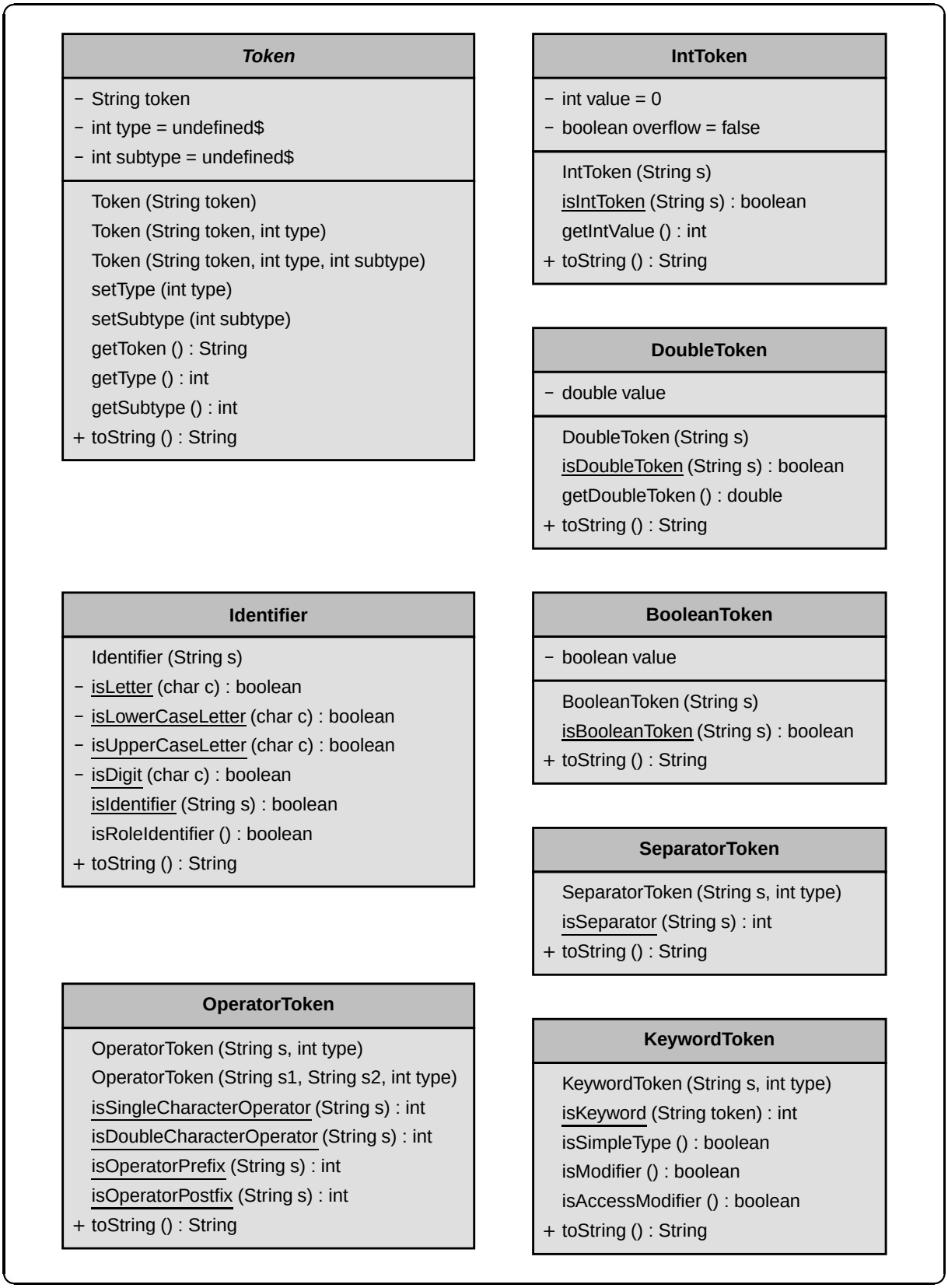


Abb. 104 Die abstrakte Klasse **Token** und ihre Unterklassen

5.10.4 Der Aufbau der Klassen RoleModelDescription, RoleDescription und InheritanceStructure

Ein Objekt der Klasse **RoleModelDescription** (vgl. Abb. 105) enthält die interne Beschreibung eines Rollenmodells. Über die Attribute `rootRelationship` und `relationship` wird die Graphrepräsentation des Rollenmodells aufgebaut. Die beiden **Vector**-Objekte `rootRelationship` und `relationship` enthalten jeweils Arrays des Typs **RoleDescription**. Jedes Array besitzt als Element mit dem Index 0 eine Oberrolle. Die folgenden Elemente stellen die direkten Unterollen dar. Ein **RoleDescription**-Array enthält daher alle Kanten des Graphen, die

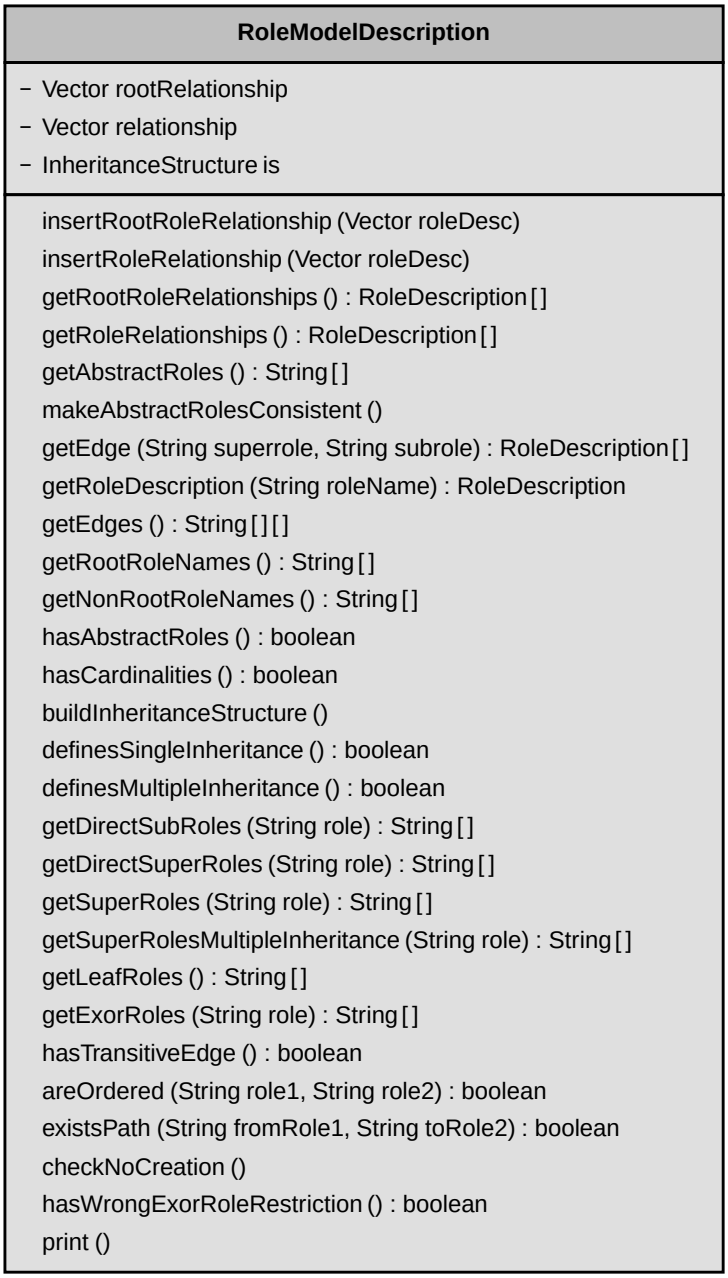


Abb. 105 Die Klasse **RoleModelDescription**

von dem Knoten wegführen, der über die Oberrolle definiert ist. Abb. 106 zeigt die Repräsentation des Rollenmodells aus Abb. 77 (S. 162). Die Rollen des Rollenmodells sind innerhalb

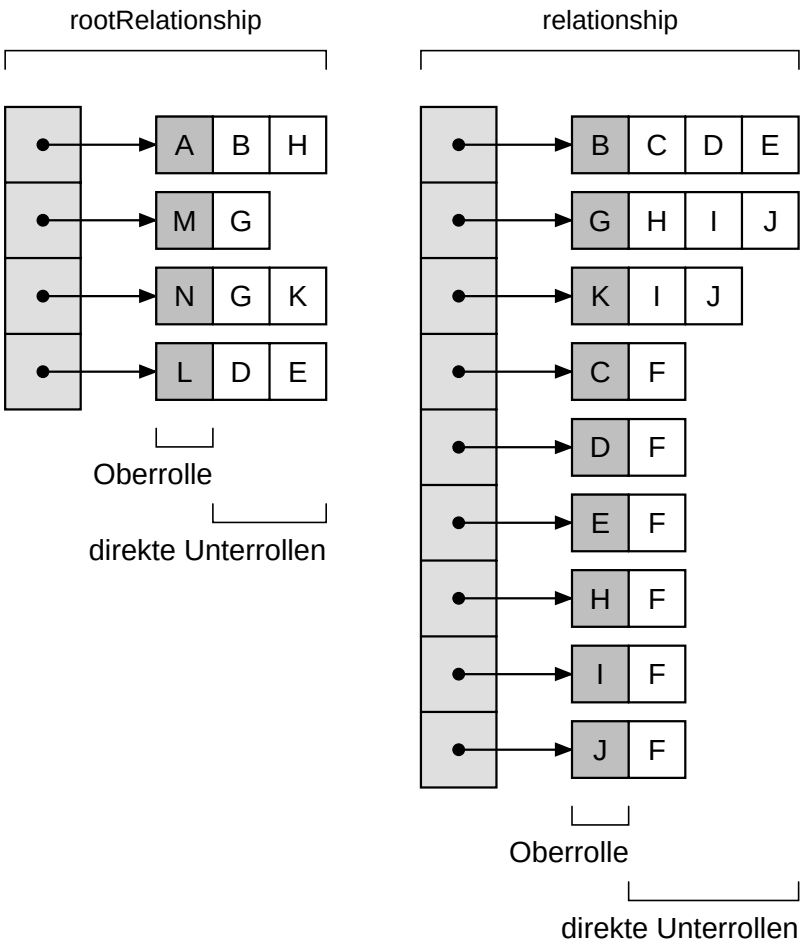


Abb. 106 Die Graphrepräsentation des Rollenmodells aus Abb. 77 (S. 162)

dieser Darstellung über **RoleDescription**-Objekte (vgl. Abb. 107 (S. 205)) beschrieben. Die Klasse besitzt Attribute, die zu einem Knoten des Graphen gehören: `name` stellt den Rollennamen dar, `isAbstract` enthält die Information, ob es sich um eine abstrakte Rolle handelt. Die restlichen Attribute beziehen sich auf die Kante des Graphen, an der das **RoleDescription**-Objekt beteiligt ist: `type` kann die Werte `rootrole$`, `superrole$` und `subrole$` annehmen. Da diese Konstanten nur innerhalb der **RoleDescription**-Klasse benötigt werden, sind sie dort lokal definiert. Die Attribute `hasExorSubroles` und `exorSubrole` sind nur dann definiert, wenn das **RoleDescription**-Objekt eine Oberrolle darstellt. Falls die Kante an einer **exor**-Restriktion beteiligt ist, enthält das Zeichenketten-Array `exorSubrole` die Namen aller Unterrollen, die zu der Restriktion gehören. Die verbleibenden Attribute sind definiert, falls `type` den Wert `subrole$` besitzt: `noCreation` legt fest, ob die Unterrolle erzeugt werden darf, `minCardinality` und `maxCardinality` enthalten die untere bzw. obere Grenze für die Unterrollenkardinalität. Bei den Methoden der Klasse **RoleDescription** handelt es sich um reine Zugriffs- und Abfragemethoden.

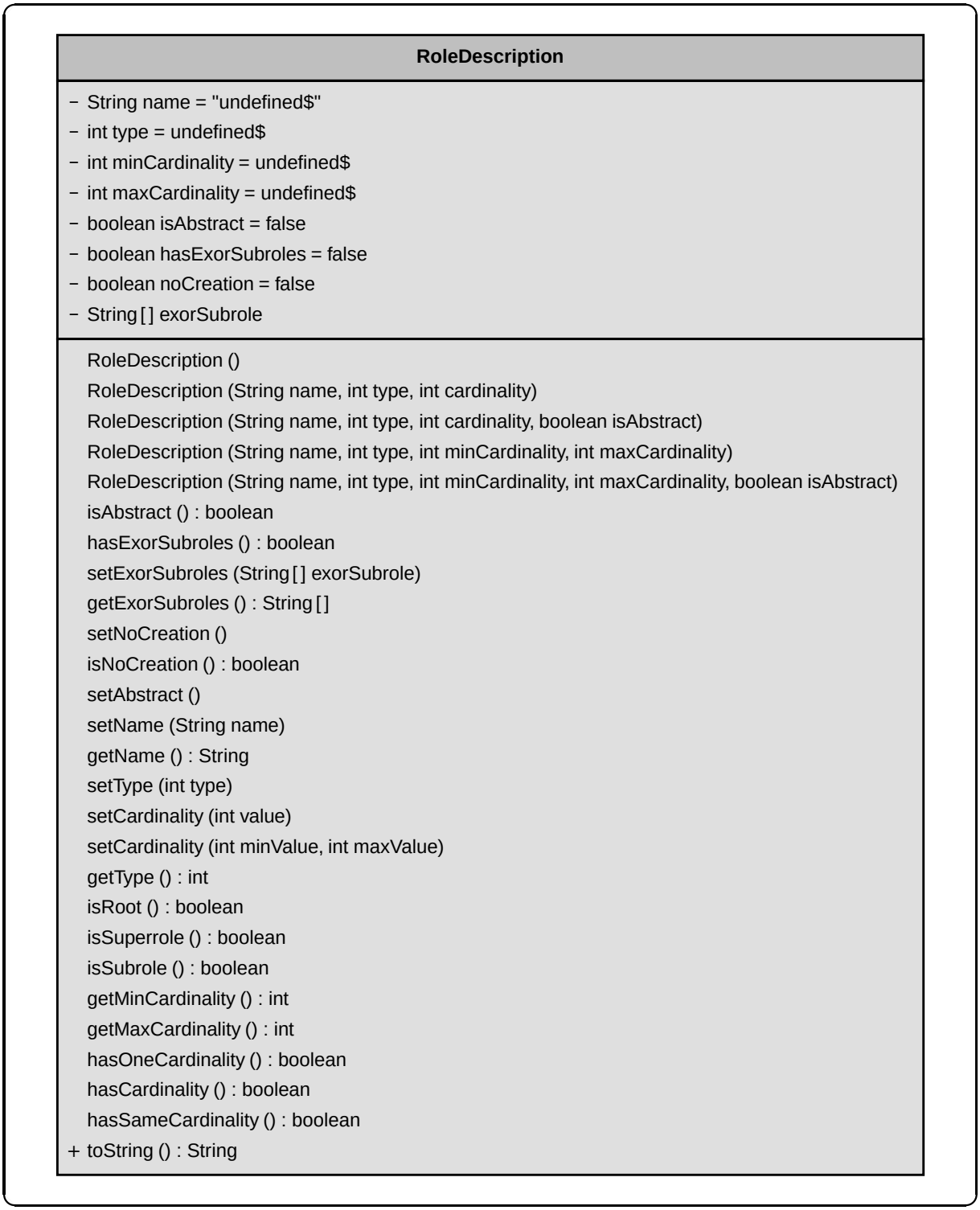


Abb. 107 Die Klasse **RoleDescription**

Beim Aufbau eines **RoleModelDescription**-Objekts wird zusätzlich intern ein **InheritanceStructure**-Objekt erzeugt (vgl. Abb. 108 (S. 206)). Dieses Objekt stellt zwei weitere Repräsentationen des Graphen zur Verfügung, die für bestimmte Operationen besser geeignet sind, als die Darstellung des Rollenmodells im **RoleModelDescription**-Objekt. Ein **InheritanceStructure**-Objekt besitzt neben einer Adjazenzmatrix eine Pfadmatrix. Das Element `am[i,j]` der

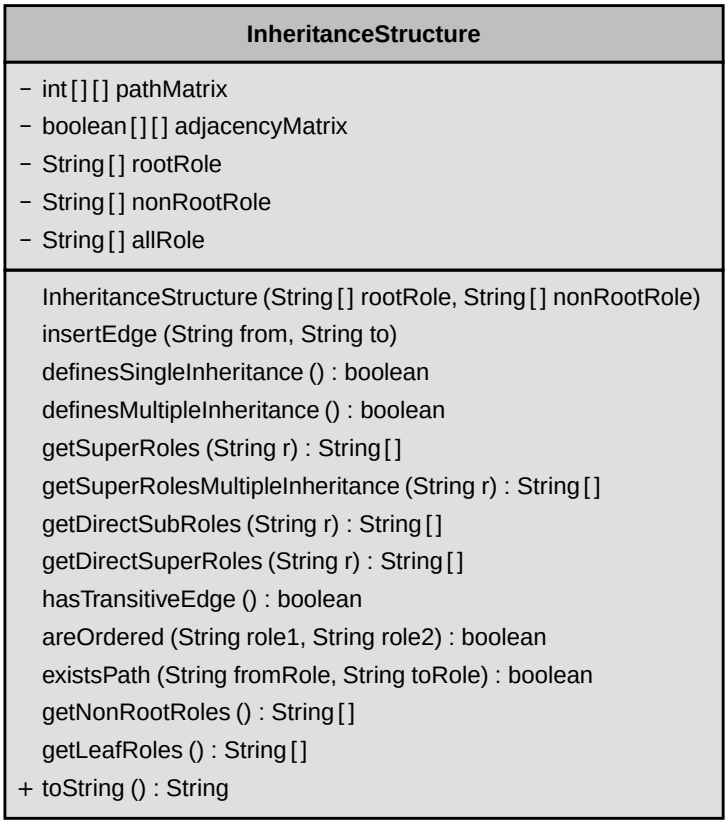


Abb. 108 Die Klasse **InheritanceStructure**

Adjazenzmatrix ist ein boolescher Wert, der festlegt, ob eine Kante vom Knoten `i` zum Knoten `j` führt oder nicht. Über die Adjazenzmatrix lassen sich z.B. sehr einfach die direkten Oberrollen einer Rolle bestimmen. Das Element der Pfadmatrix `pm[i,j]` enthält die Information, wieviele Pfade zwischen dem Knoten `i` und dem Knoten `j` existieren. Mit dieser Definition folgt: der Graph ist nicht zyklenfrei, wenn es mindestens ein Knotenpaar gibt, für das `pm[i,j]>0` und `pm[j,i]>0` gilt ($i \neq j$). Diese Eigenschaft wird beim Aufbau des **InheritanceStructure**-Objekts überprüft. Führt das Einfügen einer Kante durch die Methode `insertEdge` zu einem Zyklus, so wird eine `InheritanceCycle`-Exception erzeugt. Die Klasse stellt weitere Methoden zur Ermittlung von Strukturinformationen des Graphen zur Verfügung:

- `definesSingleInheritance`: Liegt ein Rollenmodell mit Einfachvererbung vor?
- `definesMultipleInheritance`: Liegt ein Rollenmodell mit Mehrfachvererbung vor?
- `hasTransitiveEdge`: Besitzt der Graph mindestens eine transitive Kante (vgl. S. 151)?
- `areOrdered (String role1, String role2)`: Sind die über `role1` und `role2` identifizierten Knoten geordnet oder nicht (vgl. S. 156)?
- `existsPath (String role1, String role2)`: Existiert zwischen den zugehörigen Knoten ein Pfad?
- `getSuperRolesMultipleInheritance (String r)` liefert für die Rolle `r` alle direkten und indirekten Oberrollen topologisch sortiert (vgl. S. 156) als Ergebnis.

Ein Objekt der Klasse **InheritanceStructure** ist nur für ein **RoleModelDescription**-Objekt

sichtbar, das dann nach außen alle Strukturinformationen eines Rollenmodells bereitstellt. Weitere wichtige Methoden der Klasse **RoleModelDescription** sind (vgl. Abb. 105 (S. 203)):

- `makeAbstractRolesConsistent`: In der Beschreibung eines Rollenmodells kann eine Rolle mehrmals auftreten, sowohl als Ober- wie auch als Unterolle. Die Eigenschaft, daß eine Rolle abstrakt ist, kann jedoch nur für Oberrollen spezifiziert werden (vgl. die *`RoleDeclaration1`*-Regel der *`RoleModelDescription`*-Grammatik von S. 181). Die `makeAbstractRolesConsistent`-Methode dient dazu, in allen **RoleDescription**-Objekten der Rolle das `isAbstract`-Attribut zu setzen.
- `checkNoCreation` testet die Konsistenz der spezifizierten **nocreation**-Restriktionen (vgl. Abschnitt 5.6.3 (S. 179)).
- `hasWrongExorRoleRestriction` überprüft, ob semantisch falsche **exor**-Restriktionen definiert wurden (vgl. Abschnitt 5.6.2 (S. 174)).

5.10.5 Die Klasse RoleModelParser

Ein Objekt der Klasse **RoleModelParser** aus Abb. 109 führt die Syntaxprüfung für eine Rollenmodellbeschreibung aus. Der Aufruf der `checkSyntax`-Methode liefert im Erfolgsfall ein **RoleModelDescription**-Objekt als Ergebnis. In Tab. 2 (S. 208) ist eine Zuordnung zwischen

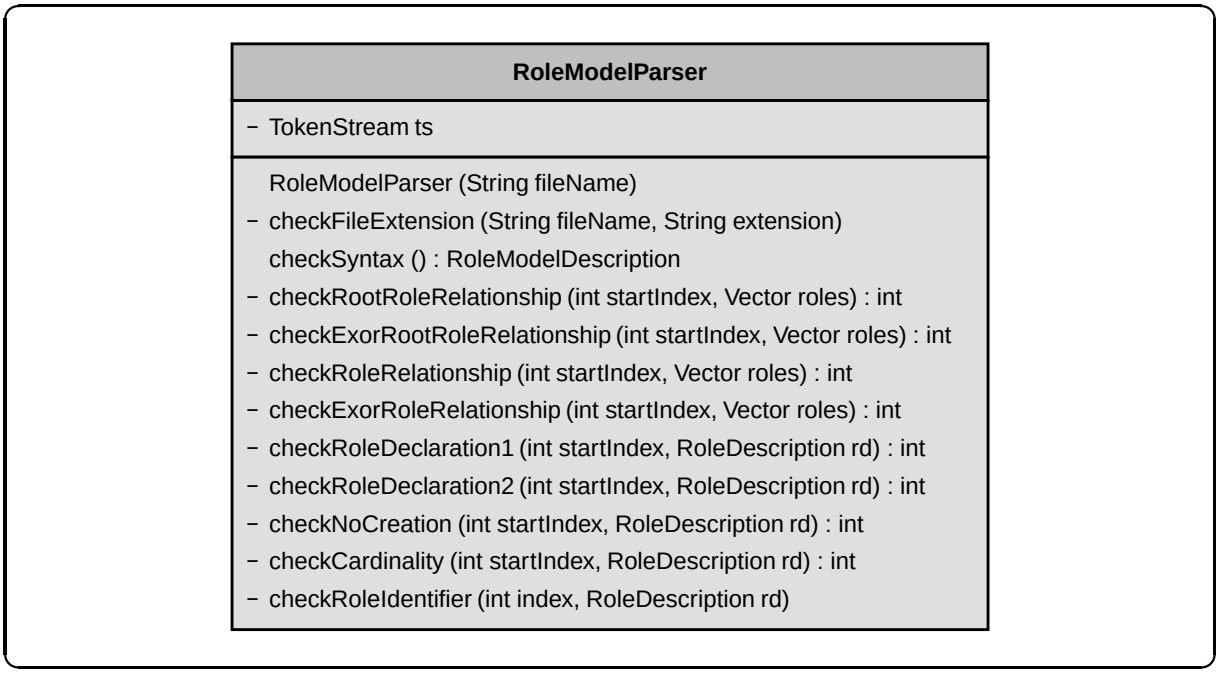


Abb. 109 Die Klasse **RoleModelParser**

den Methoden der **RoleModelParser**-Klasse und den kontrollierten Syntaxregeln der Grammatik aus Abschnitt 5.7 (S. 181) enthalten. Die Methoden `checkRootRoleRelationship`, `checkExorRootRoleRelationship`, `checkRoleRelationship` sowie `checkExorRoleRelationship` besitzen jeweils als Parameter eine Referenz auf ein **Vector**-Objekt `roles`; `roles` enthält nach der Methodenausführung das bei der Analyse der Syntaxregel neu erzeugte **RoleDescription**-Array. Dieses wird danach durch einen Aufruf von `in-`

Methode	Kontrollierte Regeln
checkSyntax	*RoleModelDescription* *RootRoleRelationships* *RoleRelationships*
checkRootRoleRelationship	*RootRoleRelationship*
checkExorRootRoleRelationship	*ExorRootRoleRelationship*
checkExorRoleRelationship	*ExorRoleRelationship*
checkRoleDeclaration1	*RoleDeclaration1*
checkRoleDeclaration2	*RoleDeclaration2*
checkCardinality	*Cardinality*
checkNoCreation	*NoCreation*
checkRoleIdentifier	*RoleIdentifier*

Tab. 2 Die Zuordnung der Methoden zu den Syntaxregeln

sertRootRoleRelationship bzw. insertRoleRelationship in das **RoleModelDescription**-Objekt integriert. Der **RoleDescription**-Parameter bei den Methoden checkRoleDeclaration1, checkRoleDeclaration2, checkNoCreation, checkCardinality und checkRoleIdentifier wird benötigt, weil diese Methoden Änderungen an den Attributwerten der zugehörigen Rollenbeschreibung vornehmen können.

5.10.6 Die Klassen ClassStructure und Declaration

Die Klasse **ClassStructure** (vgl. Abb. 110 (S. 209)) bildet die Grundlage für die Erzeugung der zu einer korrespondierenden Klasse gehörenden Rollenklasse. Der Konstruktur besitzt als Parameter den Namen der Datei, in der der Quelltext der korrespondierenden Klasse enthalten ist. Zunächst wird ein **Scanner**-Objekt erzeugt, dessen scan-Methode als Ergebnis das **Token**-Array der korrespondierenden Klasse zurückliefert. Durch den Aufruf der Methode determineDeclarations findet danach eine Zerlegung des **Token**-Arrays in die deklarierten Elemente statt. Jedes Element wird durch ein Objekt des **Declaration**-Typs (vgl. Abb. 111 (S. 210)) beschrieben. Nach der Beendigung der determineDeclarations-Methode befinden sich die Referenzen auf die **Declaration**-Objekte im **Vector**-Attribut declaration. Die Methode decomposeDeclarationVector ordnet die **Declaration**-Objekte aus declaration dann den Attributen packageDeclaration bis methodDeclaration zu.

Die abstrakte Klasse **Declaration** (vgl. Abb. 111 (S. 210)) ist die Oberklasse der Klassen **AbstractMethodDeclaration** bis **PackageDeclaration** (vgl. Abb. 112 (S. 210) und Abb. 113 (S. 211)). Das Attribut ta nimmt die Teilmenge der Symbole der korrespondierenden Klasse auf, die zu der entsprechenden Deklaration gehören. In dem type-Attribut wird der Typ der Deklaration festgehalten, wobei die in den Schnittstellen **Tokens** bzw. **Declarations** definierten Konstanten verwendet werden (vgl. Abb. 102 (S. 200)). Die Methoden der Klasse **Declaration** dienen zur Suche von Symbolen (searchToken), zum Zugriff auf die verschiedenen

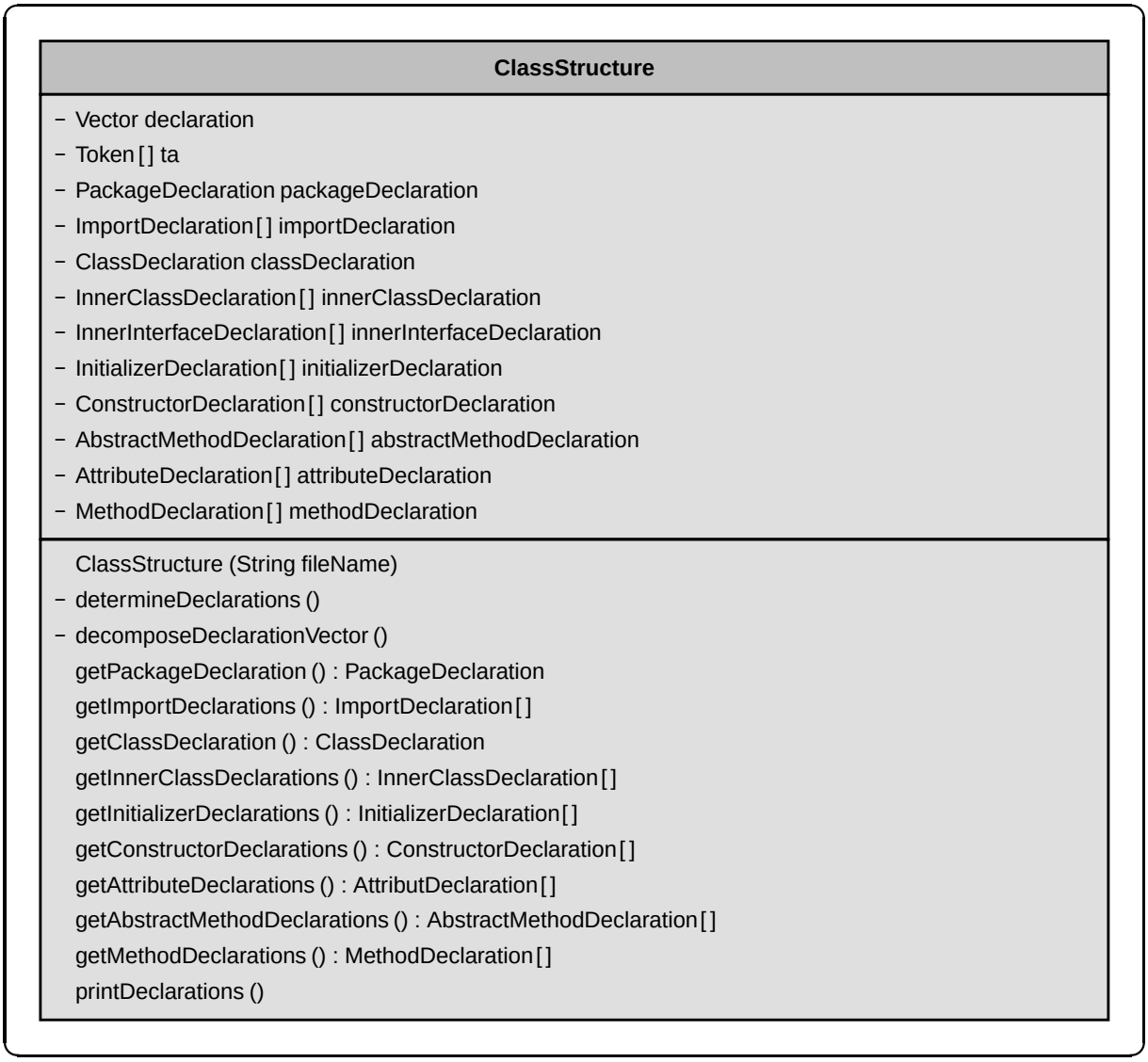


Abb. 110 Die Klasse **ClassStructure**

Modifikatoren (`getAccessModifiers`, `getNonAccessModifiers` und `writeNonAccessModifiers`) sowie zur Ermittlung eines Typnamens[25] (`getTypeName`).

Die Klasse **ClassDeclaration** (vgl. Abb. 112 (S. 210)) besitzt Methoden für das Lesen des Klassennamens (`getClassName`) und des Oberklassennamens (`getSuperClassName`).

Die abstrakte Klasse **ConstructorOrMethodDeclaration** (vgl. Abb. 112 (S. 210)) stellt Methoden zur Verfügung, die sowohl für Konstruktor- als auch für Methodendeklarationen benötigt werden. Mit diesen Methoden kann auf Parameternamen (`getParameterName`), die Liste der formalen Parameter (`getFormalParameterList`) und die Exception-Typen (`getExceptionTypes`, `writeThrowsClause`, und `writeExceptionList`) zugegriffen werden bzw. eine Verarbeitung dieser Elemente stattfinden.

Die Klasse **ConstructorDeclaration** enthält Methoden zur Abfrage, ob der Konstruktor öffentlich ist (`isPublic`), und zum Lesen des Konstruktornamens (`getConstructorName`). In

[25] Hierbei kann es sich z.B. um den Typ eines Attributs, den Ergebnistyp einer Methode oder auch einen Parametertyp handeln.

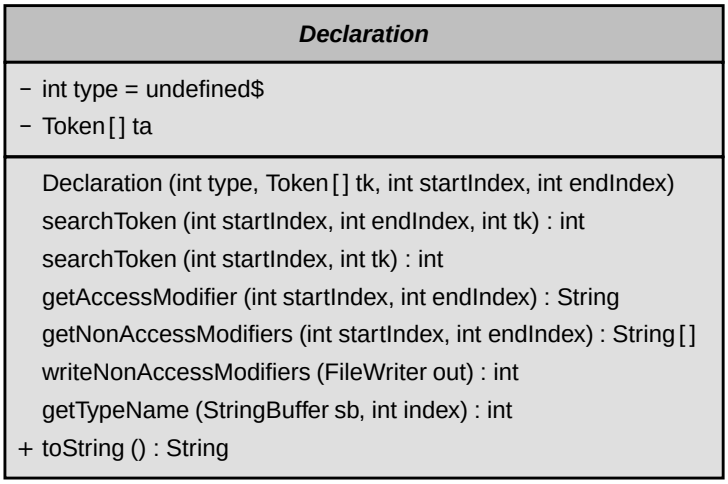


Abb. 111 Die abstrakte Klasse **Declaration**

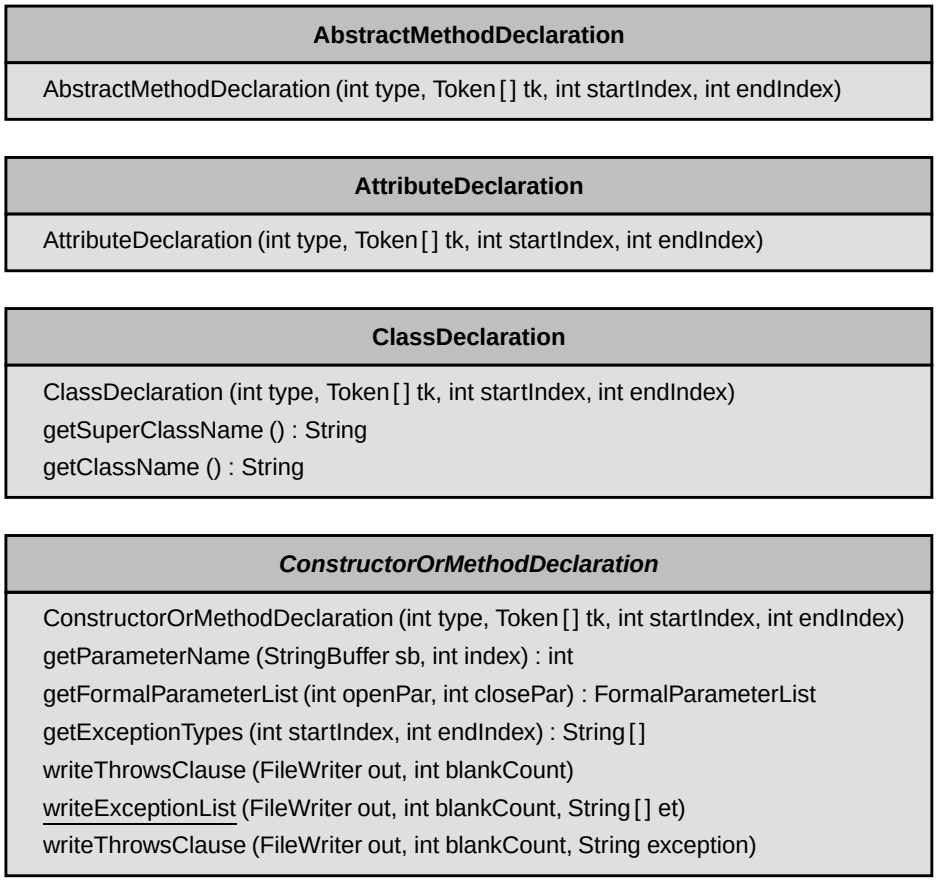


Abb. 112 Die Klassen **AbstractMethodDeclaration** bis **ConstructorOrMethodDeclaration**

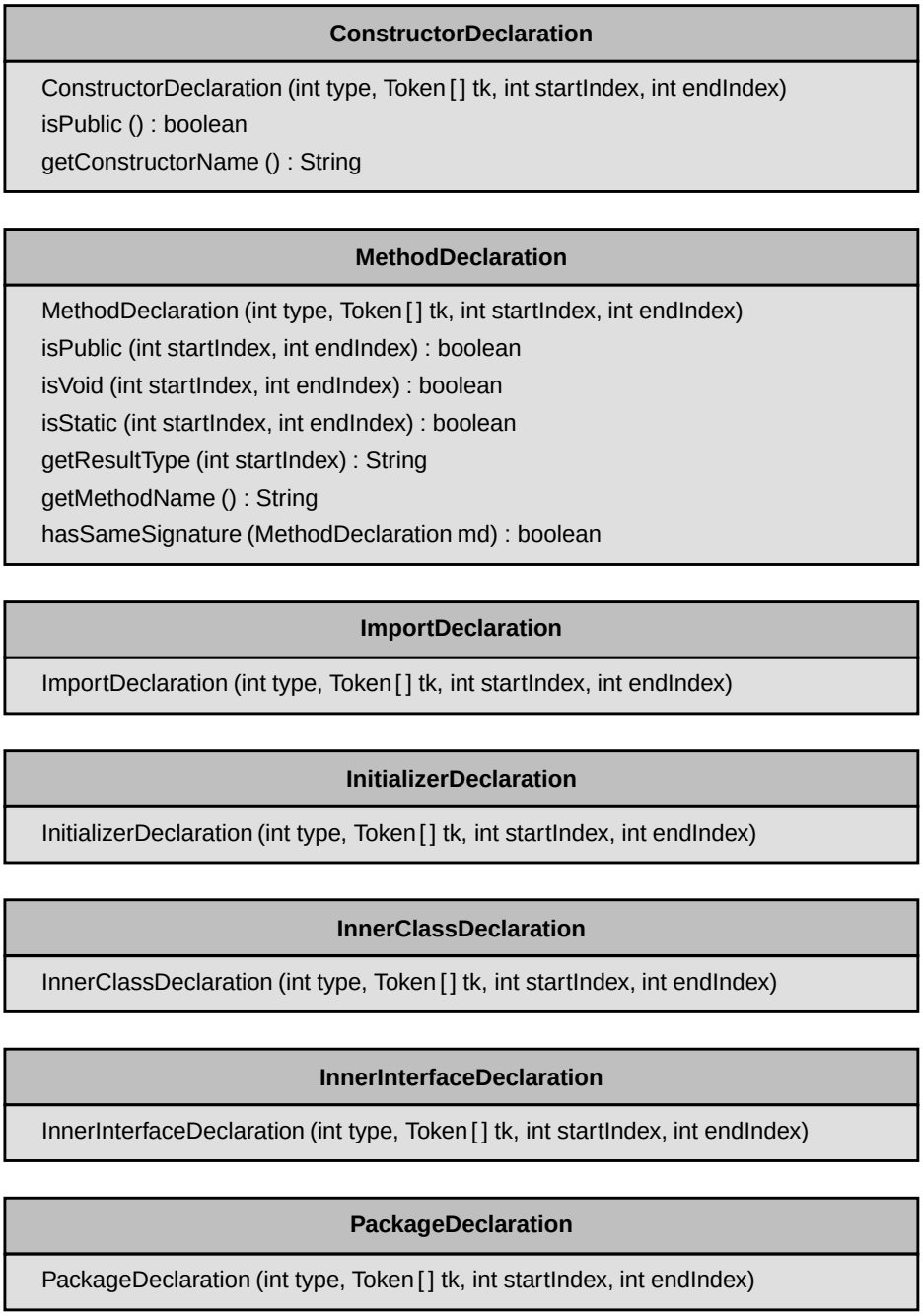


Abb. 113 Die Klassen **ConstructorDeclaration** bis **PackageDeclaration**

der Klasse **MethodDeclaration** (vgl. Abb. 113) finden sich Methoden zur Abfrage von Modifikatoren (`isPublic`, `isStatic`), des Ergebnistyps (`isVoid`, `getResultType`) und des Methodennamens (`getMethodName`). Die Methode `hasSameSignature` testet, ob zwei Methoden dieselbe Signatur besitzen, und wird für die Suche redefinierter Methoden eingesetzt.

5.10.7 Die Klassen RoleModelGenerator und ClassStructureHashtable

Die abstrakte Klasse **RoleModelGenerator** (vgl. Abb. 114) ist die Oberklasse der beiden Klassen **RoleModelSingleInheritance** und **RoleModelMultipleInheritance**, die letztlich zur Generierung eines Rollenmodells eingesetzt werden.

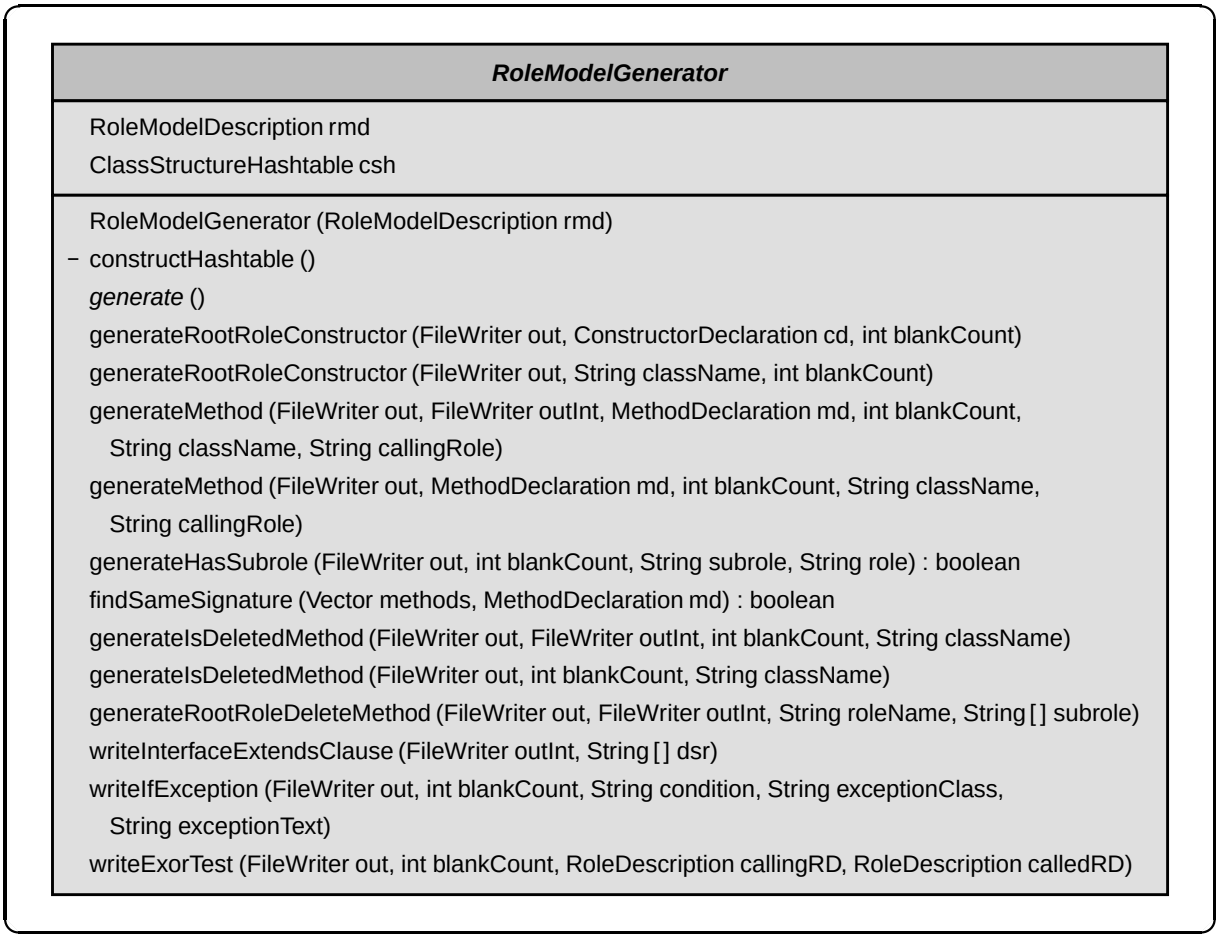


Abb. 114 Die Klasse **RoleModelGenerator**

Sie besitzt als Attribute die Rollenmodellbeschreibung (`rmd`) und eine Hash-Tabelle (siehe Abb. 115 (S. 213)) für den effizienten Zugriff auf die **ClassStructure**-Objekte. Die Methoden erfüllen die folgenden Aufgaben:

- `generateRootRoleConstructor (FileWriter out, ConstructorDeclaration cd, int blankCount)`:

  Es wird der öffentliche Konstruktor für eine Wurzelrolle des Rollenmodells erzeugt. Der Parameter `cd` beschreibt den entsprechenden Konstruktor der korrespondierenden Klasse.

  Generierungsbeispiel: Der Konstruktor `RolePerson` aus Prg. 6 (S. 113).

- `generateRootRoleConstructor (FileWriter out, String className, int blankCount)`:

  Falls die korrespondierende Klasse keinen öffentlichen Konstruktor besitzt, erzeugt die Methode einen parameterlosen Konstruktor für die Rollenklasse.

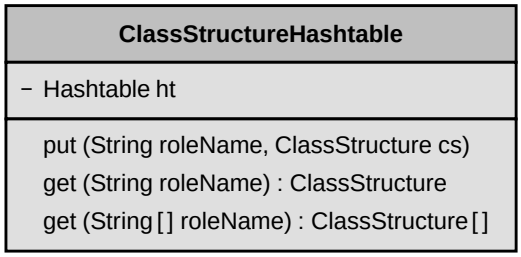


Abb. 115 Die Klasse **ClassStructureHashtable**

Generierungsbeispiel: Der Konstruktor `RoleAB` aus Prg. 22 (S. 131).

- `generateMethod (FileWriter out, MethodDeclaration md, int blankCount, String className, String callingRole):`

  Der Parameter `md` beschreibt die Methode der korrespondierenden Klasse, für die die Methode in der Rollenklasse erzeugt wird.

  Generierungsbeispiel: Die Methode `druckeName` aus Prg. 6 (S. 113).

- `generateMethod (FileWriter out, FileWriter outInt, MethodDeclaration md, int blankCount, String className, String callingRole):`

  Hier wird zusätzlich die Methodensignatur in das für die Rollenklasse erzeugte Interface aufgenommen (vgl. Abb. 98 (S. 189)).

- `generateHasSubrole (FileWriter out, int blankCount, String subrole, String role):`

  Falls für die betroffene Unterrolle `subrole` keine Kardinalität spezifiziert wurde, wird eine `hasSubrole`-Methode generiert. Liegt eine endliche Kardinalität vor, wird eine `hasAllSubroles`-Methode erzeugt.

  Generierungsbeispiel: Die `hasAllSubrolesTutor`-Methode in Prg. 27 (S. 147).

- `findSameSignature (Vector methods, MethodDeclaration md):`

  Die Methode testet, ob in der Menge `methods` mindestens eine Methode enthalten ist, die dieselbe Signatur besitzt, wie die über `md` definierte Methode. Diese Methode wird eingesetzt, um redefinierte Methoden zu finden (vgl. S. 154).

- `generateIsDeletedMethod (FileWriter out, int blankCount, String className):`

  Für die über `className` spezifizierte Rolle wird eine `isDeleted`-Methode erzeugt.

  Generierungsbeispiel: Die `isDeleted`-Methode in Prg. 8 (S. 115).

- `generateIsDeletedMethod (FileWriter out, FileWriter outInt, int blankCount, String className):`

  Hier wird zusätzlich die `isDeleted`-Signatur in das für die Rollenklasse erzeugte Interface aufgenommen (vgl. Abb. 98 (S. 189)).

- `generateRootRoleDeletedMethod (FileWriter out, FileWriter outInt, String roleName, String[] subrole):`

Für die Wurzelrolle `roleName` wird die `delete`-Methode erzeugt. `subrole` enthält die Namen der Unterrollen, für die selbst wieder ein `delete` aufzurufen ist, um neben dem Wurzelrollenobjekt auch alle Unterrollenobjekte zu löschen.

Generierungsbeispiel: Die `delete`-Methode aus Prg. 8 (S. 115).

- `writeInterfaceExtendsClause (FileWriter outInt, String[] dsr):`

  Die Methode erzeugt die `extends`-Klausel für das Interface einer Unterrolle. Dieses Interface erweitert die Interfaces aller direkten Unterrollen `dsr`.

  Generierungsbeispiel: Das Interface `RoleFInt` aus Prg. 32 (S. 173).

- `writeIfException (FileWriter out, int blankCount, String condition, String exceptionClass, String exceptionText):`

  Es wird eine `if`-Anweisung mit der Bedingung `condition` sowie einer `throws`-Anweisung generiert. Die `throws`-Anweisung erzeugt ein Exception-Objekt der Klasse `exceptionClass`, wobei der Konstruktoraufruf `exceptionText` als Parameterwert erhält.

  Generierungsbeispiel: Die Zeilen 90 bis 91 aus Prg. 8 (S. 115).

- `writeExorTest (FileWriter out, int blankCount, RoleDescription callingRD, RoleDescription calledRD):`

  Diese Methode generiert für eine `createRole`-Methode die `if`-Anweisung, die kontrolliert, ob die über `calledRD` identifizierte Unterrolle aufgrund der spezifizierten **exor**-Restriktion erzeugt werden darf.

  Generierungsbeispiel: Die Zeilen 7 bis 9 aus Prg. 34 (S. 178).

5.10.8 Die Klasse RoleModelMultipleInheritance

Ein Objekt der Klasse **RoleModelMultipleInheritance** (vgl. Abb. 116 (S. 215)) ist für die Generierung eines Rollenmodells mit Mehrfachvererbung zuständig. Die Methoden haben die folgende Bedeutung:

- `allMethodsValid():`

  Diese Methode testet für alle Unterrollen, ob unerlaubte Methodenredefinitionen vorhanden sind (vgl. S. 154). Die Methode wird einmal für das gesamte Rollenmodell ausgeführt. Bei dem Vorliegen einer fehlerhaften Methodenredefinition wird der Generierungsprozeß abgebrochen.

- `generate():`

  Die `generate`-Methode steuert den Generierungsprozeß. Zunächst werden die JAVA-Klassen für alle Wurzelrollen erzeugt, danach die JAVA-Klassen für die Unterrollen.

- `getAllPublicMethods (ClassStructure rcs):`

  Die Methode sucht alle öffentlichen Methoden der über `rcs` identifizierten korrespondierenden Klasse. Dabei werden auch alle Oberklassen berücksichtigt. Als Ergebnis liefert `getAllPublicMethods` für die gefundenen Klassen die `MethodDeclaration`-Objekte zurück.

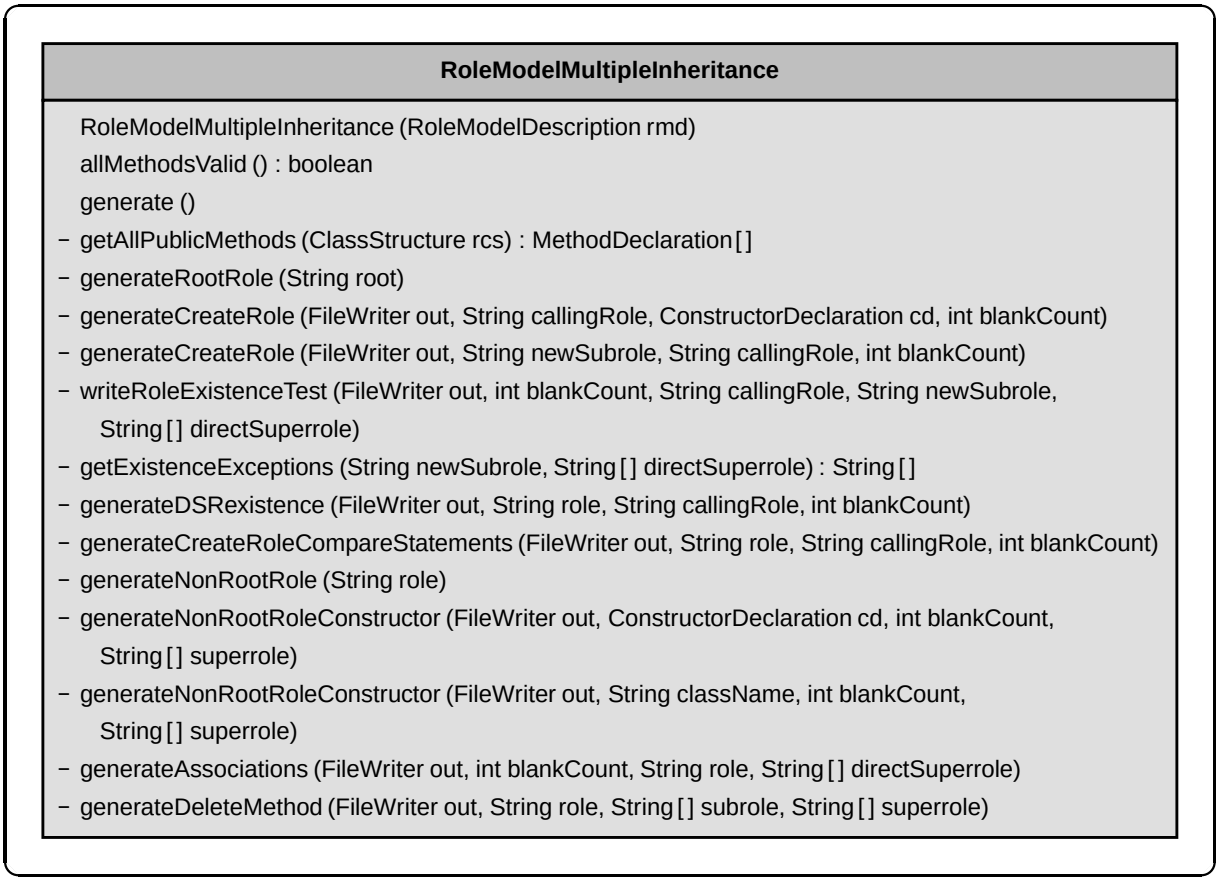


Abb. 116 Die Klasse **RoleModelMultipleInheritance**

- `generateRootRole (String root)`:

  Zunächst wird für die Wurzelrolle `root` das `ClassStructure`-Objekt ermittelt. Danach folgt die Generierung aller Attribute, Konstruktoren und Methoden der Wurzelrolle.

  Generierungsbeispiel: Die Wurzelrolle `RoleAB` aus Prg. 22 (S. 131) und Prg. 23 (S. 132).

- `generateCreateRole (FileWriter out, String callingRole, ConstructorDeclaration cd, int blankCount)`:

  Für die Oberrolle `callingRole` wird eine `createRole`-Methode generiert. Über den Parameter `cd` läßt sich ermitteln, für welche direkte Unterrolle von `callingRole` die `createRole`-Methode zu erzeugen ist.

  Generierungsbeispiel: Die `createRoleWissenschaftlicherMitarbeiter`-Methode aus Prg. 10 (S. 117).

- `generateCreateRole (FileWriter out, String newSubrole, String callingRole, int blankCount)`:

  Diese `generateCreateRole`-Variante wird verwendet, wenn eine korrespondierende Klasse keinen expliziten Konstruktor besitzt.

  Generierungsbeispiel: Die `createRoleD`-Methode aus Prg. 34 (S. 178).

- `writeRoleExistenceTest (FileWriter out, int blankCount, String callingRole, String newSubrole, String[] directSuperrole)`:

  Diese Methode generiert den Test, ob die neue Unterrolle `newSubrole` ausgehend von `callingRole` aufgrund der im Rollenmodell spezifizierten Kardinalitäten noch erzeugt werden darf (vgl. die Eigenschaft E2, S. 152). Hierfür müssen die direkten Oberrollen `directSuperrole` untersucht werden.

  Generierungsbeispiel: Die Zeilen 17 bis 28 aus Prg. 30 (S. 166).

- `getExistenceExceptions (String newSubrole, String[] directSuperrole)`:

  Die Methode liefert als Ergebnis die Menge der Exception-Klassen, die bei der Überprüfung, ob eine Unterrolle erzeugt werden darf, benötigt werden. Die konkreten Exception-Klassen hängen von den spezifizierten Kardinalitäten ab. Liegt für eine Rollenbeziehung `directSuperrole[i]` nach `newSubrole` keine Kardinalität vor, wird der Menge eine `SubroleExists`-Exception hinzugefügt; wurde dagegen eine endliche Kardinalität spezifiziert, ist eine `AllSubrolesExist`-Exception hinzuzufügen. Die Menge der Exceptions der `throws`-Klausel einer `createRole`-Methode wird dann um die hier ermittelten Exceptions ergänzt.

- `generateDSRexistence (FileWriter out, String role, String callingRole, int blankCount)`:

  Diese Methode generiert den Test, ob für die in `callingRole` neu zu erzeugende Unterrolle `role` alle benötigten direkten Oberrollenobjekte existieren (vgl. die Eigenschaft E1, S. 152).

  Generierungsbeispiel: Die Zeilen 7 bis 16 aus Prg. 30 (S. 166).

- `generateCreateRoleCompareStatements (FileWriter out, String role, String callingRole, int blankCount)`:

  Diese Methode generiert den Test, der sicherstellt, daß die Erzeugung der Unterrolle `role` nicht zu einer Konsistenzverletzung der Rollenobjektstruktur führt (vgl. die Eigenschaft E3, S. 152).

  Generierungsbeispiel: Die Zeilen 29 bis 42 aus Prg. 30 (S. 166).

- `generateNonRootRole (String role)`:

  Im ersten Schritt wird für die Unterrolle `role` das `ClassStructure`-Objekt ermittelt. Danach erfolgt die Generierung aller Attribute, Konstruktoren und Methoden der Unterrolle.

  Generierungsbeispiel: Die Rolle `RoleBB` aus Prg. 24 (S. 133) und Prg. 25 (S. 134).

- `generateNonRootRoleConstructor (FileWriter out, ConstructorDeclaration cd, int blankCount, String[] superrole)`:

  Es wird der Konstruktor für eine Unterrolle generiert. Der Parameter `cd` beschreibt den entsprechenden Konstruktor der korrespondierenden Klasse.

  Generierungsbeispiel: Der Konstruktor `RoleE` aus Prg. 29 (S. 161).

- `generateNonRootRoleConstructor (FileWriter out, String className, int blankCount, String[] superrole)`:

  Falls die korrespondierende Klasse keinen öffentlichen Konstruktor besitzt, generiert diese Methode einen parameterlosen Konstruktor für die Rollenklasse.

- `generateAssociations (FileWriter out, int blankCount, String role, String[] directSuperrole)`:

  Diese Methode generiert für einen Konstruktor der Unterrolle `role` die Beziehungen zu den korrespondierenden Objekten der direkten Oberrollen aus `directSuperrole`.

  Generierungsbeispiel: Die Zeilen 15 bis 17 aus Prg. 29 (S. 161).

- `generateDeleteMethod (FileWriter out, String role, String[] subrole, String[] superrole)`

  Diese Methode generiert für eine Unterrolle die `delete`-Methode. Neben dem Löschen der entsprechenden Referenzen werden die `delete`- bzw. `deleteAll`-Methoden für alle direkten Unterrollenobjekte aufgerufen.

  Generierungsbeispiel: Die `delete`-Methode für `RoleE` aus Prg. 31 (S. 170).

5.10.9 Die Klasse RoleModelSingleInheritance

Ein Rollenmodell mit Einfachvererbung läßt sich sowohl mit einem Aufruf der `generate`-Methode auf einem **RoleModelMultipleInheritance**-Objekt als auch auf einem **RoleModelSingleInheritance**-Objekt (vgl. Abb. 117) generieren. Die Verwendung eines **RoleModelSingleInheritance**-Objekts ist allerdings effizienter, da die dort implementierten Methoden an die deutlich einfachere Struktur eines Rollenmodells mit Einfachvererbung angepaßt sind.

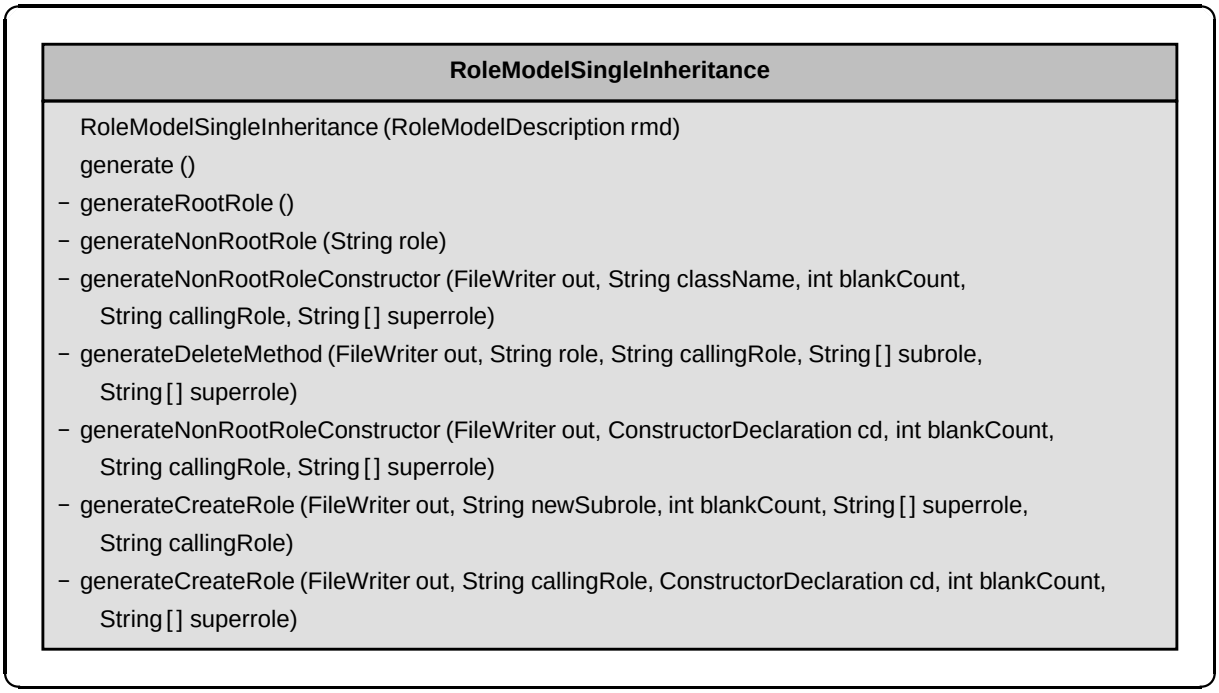


Abb. 117 Die Klasse **RoleModelSingleInheritance**

5.11 Kapitelzusammenfassung

Unter Verwendung der Definitionen und Abkürzungen aus Abschnitt 4.8 (S. 89) ist das neue Rollenmodell wie folgt zu charakterisieren:

- Struktur: G(n)
- Mehrfachrollen: M(n,m)
- Dynamik: ja
- Identität: ZR,E,L
- Redefinition: V
- Restriktionen: ja
- Migration: ja (auf der Konzeptebene)
- Typsicherheit: ja

Das neue Rollenmodell besitzt als zentrale Eigenschaft eine klare Trennung zwischen der potentiellen Struktur eines Gesamtobjekts und der Spezifikation der Funktionalität der einzelnen Rollen. Für die Beschreibung einer Rollenmodellstruktur wurde eine Grammatik definiert, über die sich die folgenden Eigenschaften festlegen lassen:

- Welche direkten Unterrollen darf eine Rolle besitzen?
  Auf diesem Weg läßt sich für ein Rollenmodell ein azyklischer Graph definieren, der mehrere Wurzelrollen enthält.
- Wieviele Rollenobjekte dürfen von einer Unterrolle gleichzeitig existieren?
- Welche Rollen sind abstrakt?
  Über eine abstrakte Rolle kann den Unterrollen eine bestimmte Funktionalität zur Verfügung gestellt werden. Im Unterschied zu einer abstrakten Klasse ist es möglich, von einer abstrakten Rolle ein Rollenobjekt zu erzeugen. Dieses besitzt aber nur Operationen zur Verwaltung von Unterrollen.
- Zwischen welchen Rollen bestehen welche Restriktionen?
  Als Restriktionen stehen die **and**-Restriktion, die **or**-Restriktion, die **exor**-Restriktion und die **nocreation**-Restriktion zur Verfügung. Die ersten beiden Restriktionen werden implizit über die Struktur des Rollenmodells definiert, während die letzten beiden Restriktionen explizit vom Anwender anzugeben sind.

Die eigentliche Funktionalität einer Rolle wird über ihre korrespondierende Klasse festgelegt. Durch einen Generierungsprozeß werden auf der Basis der Strukturbeschreibung und der korrespondierenden Klassen die Rollenklassen erzeugt, die später von einer Applikation verwendet werden. Als Programmiersprache für die Implementierung der korrespondierenden Klassen und für die erzeugten Rollenklassen wurde JAVA verwendet. Obwohl JAVA auf der Klassenebene nur Einfachvererbung erlaubt, ist durch den gewählten Ansatz die Definition und Implementierung von Rollenmodellen mit Mehrfachvererbung möglich.

Um die Konsistenz zwischen der Modellierungs- und der Implementierungsebene sicherzustellen, mußten verschiedene Testkriterien entwickelt werden. Diese Tests werden automatisch in die generierten Rollenklassen integriert, so daß es nicht dem Anwendungsprogrammierer überlassen bleibt, sich beispielsweise um die Einhaltung von Restriktionen zu kümmern. Die

Tests sind entscheidend für die Frage, ob eine Unterrolle erzeugt werden darf:

- Existieren alle benötigten direkten Oberrollenobjekte?
- Kann durch die Erzeugung der Unterrolle mit den angegebenen direkten Oberrollenobjekten die vorgegebene Struktur des Rollenmodells verletzt werden?
- Ist aufgrund der spezifizierten Kardinalitäten die Unterollenerzeugung überhaupt noch erlaubt?
- Werden Restriktionen verletzt?

Die Ermittlung der konkreten Testkriterien ist für komplexe Rollenmodelle teilweise sehr aufwendig und damit fehleranfällig. Daher ist es wichtig, diese Tests automatisch zu erzeugen.

Als weitere wichtige Eigenschaften des vorgestellten Ansatzes lassen sich nennen:

- Die Trennung zwischen der Struktur- und der Funktionalitätsdefinition entspricht dem Konzept der aspektorientierten Programmierung. Die spezifizierte Beschreibungssprache für die Rollenmodellstruktur kann als Aspektbeschreibungssprache aufgefaßt werden. Hier sind alle wesentlichen Informationen an einer Stelle zusammengefaßt, die den *Aspekt der Rollenmodellstruktur* betreffen.
- Die korrespondierenden Klassen können in unterschiedlichen Rollenmodellen wiederverwendet werden, da sie selbst keine Informationen darüber enthalten, welche Stellung sie in einem Rollenmodell haben. Damit ist es beispielsweise auch möglich, auf effiziente Weise unterschiedliche Varianten eines Rollenmodells zu erzeugen. Diese Eigenschaft ist für mobile Agentensysteme interessant, wo nicht auf jedem Rechnerknoten die vollständige Rollenmodellfunktionalität benötigt wird. Insbesondere wenn mobile Endgeräte mit einem beschränkten Speicherplatz vorliegen, ist es sinnvoll, nur die tatsächlich benötigte Funktionalität bereitzustellen.
- Durch die für die einzelnen Rollenklassen erzeugten Verwaltungsoperationen wird eine Gesamtobjektidentität gewährleistet. Obwohl intern das Gesamtobjekt aus mehreren Einzelobjekten besteht, tritt das Problem der Objektschizophrenie nicht auf, da die Verwaltungsoperationen keine Inkonsistenzen des Gesamtobjekts zulassen. Für den Anwender wird damit immer eine konsistente Gesamtobjektsicht garantiert. Auch das in [LB02] beschriebene Problem der *Role-Binding*-Anomalien kann durch die gewählte Struktur nicht auftreten.
- Es wird eine vollständige Typsicherheit erreicht.
- Für die Verwaltungsoperationen wurden entsprechende Exception-Klassen definiert. Dadurch wird der Anwender gezwungen, explizit auf Fehler zu reagieren, was die Software-Qualität erhöht.
- Unter der Annahme, daß für die korrespondierenden Klassen eine vollständige Kapselung vorliegt, ist es ohne großen Aufwand möglich, in dem Modell den Austausch der korrespondierenden Objekte durch eine neue Implementierung zur Laufzeit zu unterstützen.
- Das Modell läßt sich einfach um die Funktionalität der Rollenmigration erweitern.

Nach der Vorstellung der Eigenschaften des neuen Rollenmodells wurde beschrieben, wie sich dieses in den übergeordneten Software-Entwicklungsprozeß integrieren läßt. Den Abschluß des Kapitels stellte die Beschreibung der Software-Architektur zur Generierung des Rollenmodells dar, die in JAVA realisiert wurde. Alle in diesem Kapitel vorgestellten Implementierungen von Rollenklassen wurden damit erzeugt.

Kapitel 6

Anforderungen an ein Variantenkonzept

6.1 Einleitung

Bei der Rollenmodellierung stehen Objekte im Vordergrund, die aufgrund ihrer Langlebigkeit neue Aufgaben (Rollen) übernehmen können. Das Rollenkonzept ist sehr stark mit der *ist-ein*-Beziehung verknüpft. Die Dynamik entsteht durch die Übernahme und das Aufgeben von Rollen während des Objektlebenszyklus. Betrachtet man dagegen Objekte, deren Struktur auf einer Kompositionsbeziehung aufbaut, so paßt der Rollenbegriff an dieser Stelle nicht mehr. Ein Schrank besteht zwar (unter anderem) aus einzelnen Schubladen, das Einsetzen einer Schublade entspricht aber sicherlich nicht der Übernahme der *Rolle Schublade* durch den Schrank. Trotzdem unterliegen langlebige, zusammengesetzte Objekte auch einer Dynamik, die sich über eine Variantenbildung beschreiben läßt. Ausgehend von einem Grundobjekt können durch den Ein- und Ausbau unterschiedlicher Komponenten sehr viele Objektvarianten entstehen. In diesem Kapitel wird an dem Beispiel eines Autos ermittelt, welche Anforderungen sich an ein Variantenmodell stellen lassen.

6.2 Modellierung mit Varianten

Will man heute ein neues Auto kaufen, so besteht für ein konkretes Modell die Auswahl zwischen nahezu beliebig vielen Ausstattungsvarianten. Wird das Auto als Klasse modelliert, können viele dieser Varianten direkt über Attributwerte abgebildet werden. Als Beispiele sind hier die Leistung, der Hubraum, die Außenfarbe und die Farbe der Innenausstattung zu nennen. Diese Ausstattungsvarianten haben allerdings keinen Einfluß auf die Funktionalität des Autos, d.h. im Sinne der Objektorientierung auf die Menge der Operationen, die auf einem Autoobjekt ausführbar sind. Eine andere Situation liegt vor, wenn die Variante durch den Einbau eines bestimmten Bauteils entsteht. Wählt man als Sonderausstattung z.B. Schiebedach und Radio, so besitzt das Auto als Gesamtobjekt jetzt neue Operationen, wie das Öffnen und Schließen des Schiebedachs, das Ein- und Ausschalten des Radios, das Ändern der Lautstärke und das Wechseln des Senders. Das Vorhandensein des Radios und des Schiebedachs erhöht

aber nicht nur die Anzahl der Funktionen des Fahrzeugs, es ändern sich auch bestimmte, bereits vorhandene Operationen. Das Abschließen der Tür hat ein automatisches Schließen des Schiebedachs und ein Abstellen des Radios zur Folge. Eine weitere interessante Eigenschaft ist, daß dasselbe Bauteil unterschiedliche Operationen an seiner Schnittstelle anbieten kann, je nachdem in welches übergeordnete Bauteil es eingebaut ist. Ein Beispiel ist ein Autoradio, welches über einen PIN-Code[1]-Code geschützt wird. Wurde beim Einschalten mehrmals der falsche Code eingegeben, wird das Radio gesperrt. Ein Entsperren ist innerhalb des Autos nicht möglich. Stattdessen muß man das Radio ausbauen und zum Hersteller einsenden. Dieser baut das Radio dann in ein Testgerät ein, über das ein Entsperren unter Verwendung einer Master-PIN, die fest im Radio eincodiert ist, vorgenommen werden kann.

6.3 Anforderungen an ein Variantenmodell

Aus dem obigen Beispiel lassen sich die folgenden Anforderungen ableiten. Dabei werden die Begriffe **Chassis** für das übergeordnete Bauteil und **Komponente** (*Component*) für ein untergeordnetes Bauteil verwendet.

- AV1: Es muß die hierarchische Konstruktion von Bauteilen unterstützt werden.
  In ein Chassis können unterschiedliche Komponenten eingebaut und eventuell auch wieder ausgebaut werden.
- AV2: Eine Komponente, die in ein Chassis einbaubar ist, kann selbst wieder aus untergeordneten Bauteilen bestehen, d.h. ebenfalls die Chassis-Funktionalität übernehmen.
- AV3: Eine Komponente kann physikalisch maximal Bestandteil eines Chassis sein.
- AV4: Die aktuell verfügbare Funktionalität (d.h. die Menge der nutzbaren Operationen) einer Komponente kann davon abhängen, in welches Chassis sie eingebaut ist.
- AV5: Die Funktionalität einer Chassis-Operation kann von dem Vorhandensein bestimmter Komponenten oder Komponentenkombinationen abhängen.
- AV6: Die Funktionalität einer Komponentenoperation kann von dem Vorhandensein anderer Komponenten oder Komponentenkombinationen abhängen.

Zusätzlich zu diesen Anforderungen ist es wichtig, die Variantenmodellierung vollständig in den Systementwicklungsprozeß zu integrieren. Wie bei den Rollenmodellen muß die Konsistenz zwischen dem Analyse-, Entwurfs- und Implementierungsmodell sichergestellt werden.

6.4 Kapitelzusammenfassung

In diesem Kapitel wurden nach einer allgemeinen Abgrenzung von Rollen- und Variantenmodellen die konkreten Anforderungen an ein Variantenmodell beschrieben. Um eine möglichst weitgehende Integration der Phasen Analyse, Entwurf und Implementierung zu erreichen, sollte es möglich sein, ausgehend von einer abstrakten Beschreibung des Variantenmodells auf der Analyseebene die notwendigen Strukturen der Implementierungsebene automatisch zu erzeugen.

[1] PIN: Personal Identification Number

Kapitel 7

Existierende Ansätze zur Unterstützung der Variantenbildung

7.1 Einleitung

Die Konzepte zur Variantenbildung sind eng mit Kompositionstechniken verbunden. Sie lassen sich in zwei Kategorien unterteilen. Die erste Kategorie baut auf der ***ist-ein-Beziehung*** auf und verwendet unterschiedliche Formen des Vererbungskonzepts, um neue Klassenvarianten auf der Grundlage bereits existierender Klassen zu erzeugen. Eine Anpassung des Kontrollflusses ist hier über die Methodenredefinition möglich. Die zweite Kategorie besitzt als Ausgangspunkt die ***besteht-aus-Beziehung***. Durch ein Zusammensetzen von Objekten entstehen übergeordnete Objekte, wobei das übergeordnete Objekt die Infrastruktur für seine Teilobjekte bereitstellt. Verwendet man diesen Ansatz zur Erstellung kompletter Anwendungssysteme, so führt dies zu komponentenbasierten Systemstrukturen. Die übergeordneten Objekte werden jetzt typischerweise als Container bezeichnet und bieten ihren Objekten (Komponenten) Dienstleistungen wie z.B. eine Transaktionsverwaltung an. Die in diesem Kapitel vorgestellten Ansätze lassen sich den beiden Kategorien wie folgt zuordnen:

- Ansätze auf der Grundlage der ***besteht-aus-Beziehung***:
  - Die Kompositionsbeziehung (Abschnitt 7.2 (S. 224)).
  - Das Composite-Muster (Abschnitt 7.3 (S. 225)).
  - Die Hyperspaces-Methode und HYPER/J (Abschnitt 7.7 (S. 240)).
  - Die DEMETER-Methode und DJ (Abschnitt 7.8 (S. 243)).
  - Komponenten (Abschnitt 7.9 (S. 245)).
- Ansätze auf der Grundlage der ***ist-ein-Beziehung***:
  - Die Einfach- und Mehrfachvererbung (Abschnitt 7.4 (S. 226)).
  - Das MIXIN- und das TRAITS-Konzept (Abschnitt 7.5 (S. 231)).
  - Das RONDO-Modell (Abschnitt 7.6 (S. 236)).

Bei dieser Zuordnung ist zu beachten, daß sie die wesentliche Grundlage des jeweiligen Ansatzes herausstellen soll und keinen ausschließenden Charakter hat. Verwendet man beispielsweise zur Systemstrukturierung einen Komponentenansatz, so kann natürlich das Vererbungskonzept zur Definition der Komponenten eingesetzt werden.

7.2 Die Kompositionsbeziehung

Auf der Analyseebene bildet die Kompositionsbeziehung die Grundlage für den Aufbau hierarchischer Objektstrukturen, die für Variantenmodelle typisch sind. In Abb. 118 ist ein Ausschnitt des Vararіantenmodells eines Autos dargestellt. Die **exor**-Restriktion wird verwendet um sicherzustellen, daß ein Radio nicht gleichzeitig in einem Radiotester und einem Auto eingebaut sein kann.

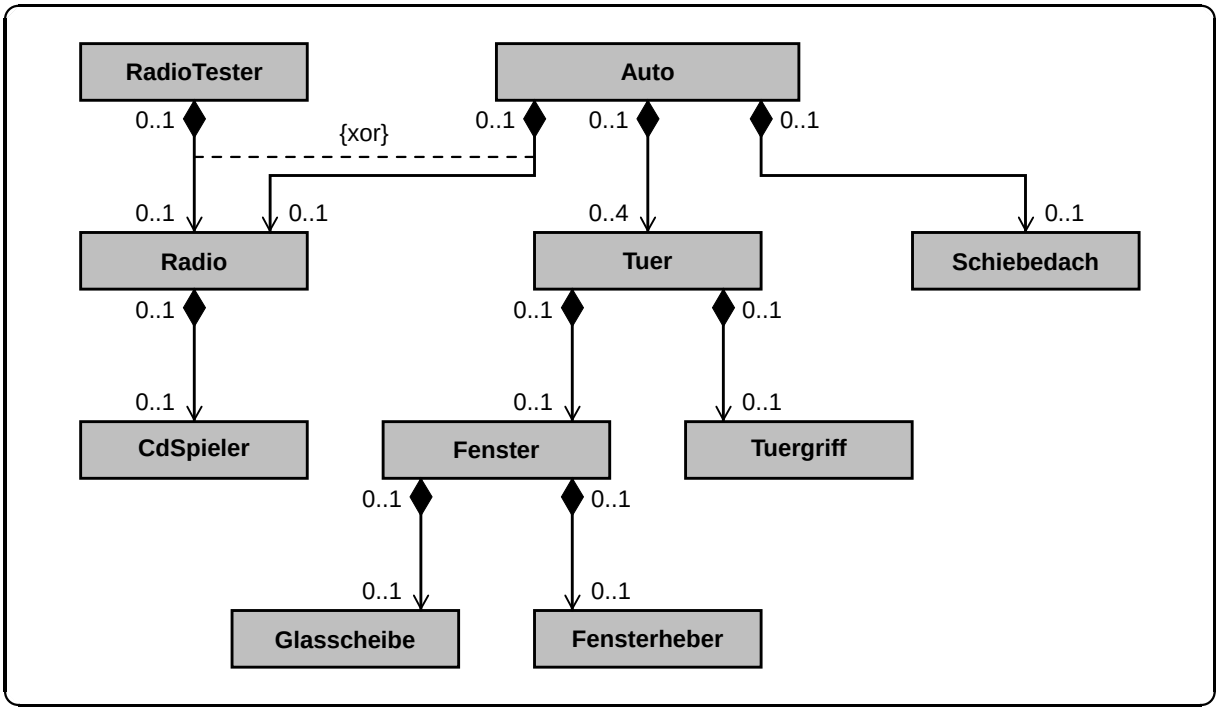


Abb. 118 Die Verwendung der Kompositionsbeziehung

Bezogen auf die UML-Spezifikation gelten für die Kompositionsbeziehung die folgenden Eigenschaften ([Bal99, S. 47]):

- Ein Objekt der Teilklasse (*Part Class*) kann höchstens in einem Objekt der Aggregatklasse (*Aggregate Class*, *Composite Class*) enthalten sein.
- Die dynamische Semantik des Aggregatobjekts gilt auch für seine Teilobjekte. Für das Beispiel aus Abb. 118 heißt dieses beispielsweise, daß aus einer Bewegung des Autos auch die Bewegung des Radios folgt.
- Wird das Aggregatobjekt gelöscht, so sind davon auch alle Teilobjekte betroffen: *They live and die with it*. Allerdings darf ein Teilobjekt vorher aus dem Aggregatobjekt entfernt werden.

Damit lassen sich auf der Analysesebene die Anforderungen AV1 bis AV3 (vgl. Abschnitt 6.3 (S. 222)) direkt erfüllen. Bei einer Umsetzung des Analysemodells in eine Implementierung ist darauf zu achten, daß die oben genannten Eigenschaften einer Kompositionsbeziehung auch eingehalten werden. Die Anforderungen AV4 bis AV6 sind jedoch durch die Verwendung einer Kompositionsbeziehung nicht automatisch erfüllt. Die Tatsache, daß zwischen zwei Objekten eine Kompositionsbeziehung besteht, hat keinen Einfluß auf die Semantik der Operationen der

beiden Objekte.[1] Aus dem Abschließen der Fahrertür folgt nicht automatisch ein Ausschalten des Radios, nur weil in dem Auto aktuell ein Radio eingebaut ist.

Zusammenfassend betrachtet kann die Kompositionsbeziehung als Grundlage für das statische Analysemodell eines Variantenmodells verwendet werden, für das dynamische Modell leistet sie jedoch keinen direkten Beitrag.

7.3 Das Composite-Muster

Das Composite-Muster wird verwendet, um über eine Baumstruktur *Whole-Part*-Hierarchien zu realisieren ([GHJV95, S. 163-173]). Es benutzt die Kompositionsbeziehung, definiert zusätzlich aber auch Zugriffs- und Verwaltungsoperationen. Die Grundstruktur des Musters ist in Abb. 119 dargestellt[2].

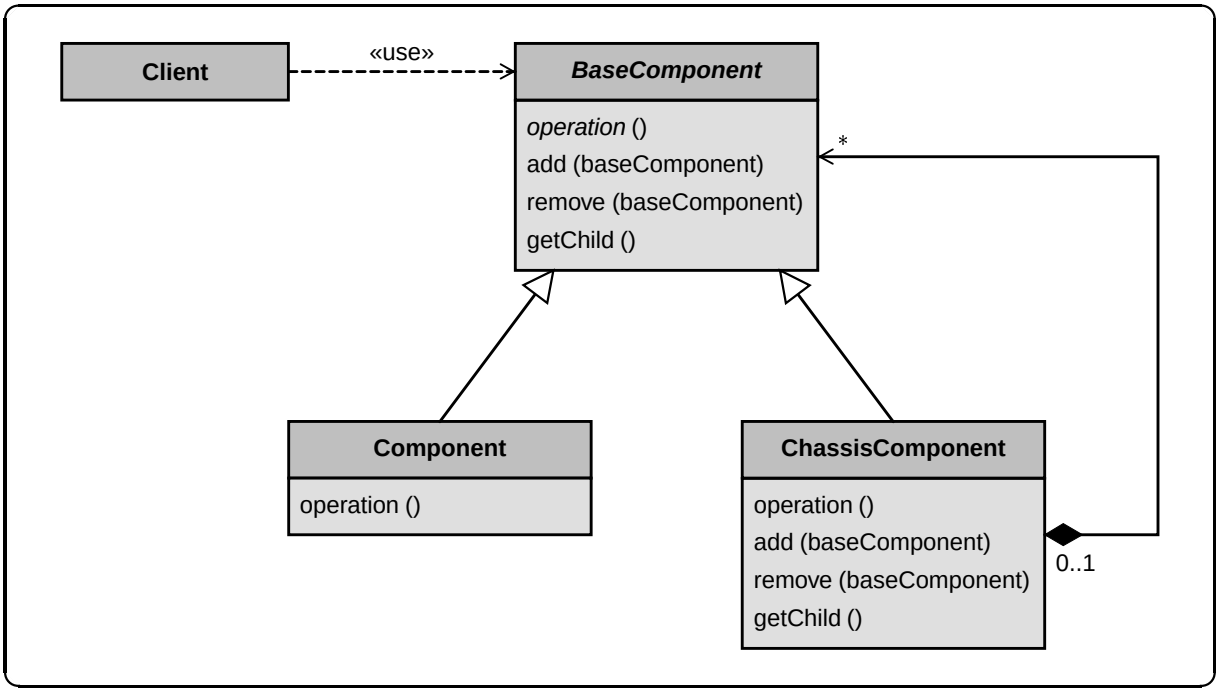


Abb. 119 Die Struktur des Composite-Musters

Die Klasse **Component** repräsentiert die Blätter der Baumstruktur, während **ChassisComponent** für die Wurzel und innere Knoten zu verwenden ist. Die Schnittstelle zu einer Anwendungsklasse (in Abb. 119 **Client**) bildet die abstrakte Klasse **BaseComponent**. Sie enthält drei Operationstypen:

- Operationen, die die Objektsemantik spezifizieren:
  Ein Beispiel wäre eine `draw`-Operation, wenn das Composite-Muster zur Beschreibung

[1] Die einzige Ausnahme ist die Löschoperation. Die oben erwähnte dynamische Semantik dient auf der Analyseebene der Abgrenzung gegenüber einer Aggregations- oder reinen Assoziationsbeziehung. Sie hat aber keinen direkten Einfluß auf die Struktur der Operationen.

[2] Die Klassennamen wurden an die hier verwendete Bezeichnungsweise angepaßt. Es gilt die folgende Zuordnung: **Component** → **BaseComponent**, **Leaf** → **Component** und **Composite** → **ChassisComponent**.

von Grafikkomponenten verwendet wird. In Abb. 119 (S. 225) ist die abstrakte Operation `operation` der Stellvertreter für diesen Operationstyp.

- Zugriffsoperationen:
  Die Operation `getChild` liefert die Referenzen aller direkten, untergeordneten Komponentenobjekte zurück.
- Verwaltungsoperationen:
  `add` und `remove` dienen zum Einfügen bzw. Entfernen von (direkten) Unterkomponenten.

Die in Abb. 119 (S. 225) präsentierte Form des Composite-Musters hat zum Ziel, den Anwenderklassen eine möglichst einheitliche Sichtweise auf die Objekte der Hierarchie zu gewähren. Daher sind die Zugriffs- und Verwaltungsoperationen der abstrakten Klasse **BaseComponent** zugeordnet, obwohl die Ausführung für **Component**-Objekte nicht sinnvoll ist. Dies sollte bei einer Implementierung der Operationen innerhalb der Klasse **BaseComponent** berücksichtigt werden. In [GHJV95, S. 167] wird deshalb empfohlen, daß `getChild` standardmäßig die leere Referenzenmenge zurückliefert[3], während `add` und `remove` mit einer Fehlermeldung reagieren. In der Klasse **ChassisComponent** wird dieses Verhalten dann redefiniert, da für **ChassisComponent**-Objekte die Operationen sinnvoll sind. Eine alternative Modellierung des Composite-Musters besteht darin, die Zugriffs- und Verwaltungsoperationen erst in der Klasse **ChassisComponent** zu definieren.

Für Operationen des Typs `operation` wird die Implementierung in der Regel die folgende Struktur besitzen:

- In einer **Component**-Klasse:
  Die eigentliche Semantik der Operation wird implementiert, also z.B. im Falle eines Grafikobjekts der Algorithmus zum Zeichnen des Objekts auf dem Bildschirm.
- In einer **ChassisComponent**-Klasse:
  Hier steht die *Container*-Eigenschaft von **ChassisComponent** im Vordergrund. Die Aufgabe von `operation` liegt darin, auf allen (direkten) Unterkomponenten wieder `operation` aufzurufen.

Da das Composite-Muster für allgemeine *Whole-Part*-Hierarchien formuliert ist, kann es zwar die ersten drei Anforderungen aus Abschnitt 6.3 (S. 222) abdecken, die Anforderungen AV4 bis AV6 sind aber nicht erfüllbar. Die Menge der nutzbaren Operationen eines Komponentenobjekts kann nicht davon abhängig gemacht werden, in welches Chassis-Objekt es aktuell eingebaut ist (Anforderung AV4). Ebenso hängt die Semantik einer Chassis- oder Komponentenoperation nicht davon ab, welche Gesamtkonfiguration gerade vorliegt (Anforderungen AV5 und AV6). Der durch das Muster unterstützte Kontrollfluß einer Chassis-Operation `operation` sieht nur den Zugriff auf die entsprechenden Operationen aller direkten Unterkomponentenobjekte vor.

7.4 Das Vererbungskonzept

Das Vererbungskonzept dient auf der konzeptuellen Ebene der Modellierung einer ***ist-ein***-Beziehung zwischen einer Unter- und einer Oberklasse. Damit kann die Unterklasse im weite-

[3] Bei einer JAVA-Implementierung kann man hierfür den Ergebniswert `null` verwenden.

sten Sinne als eine Variante der Oberklasse aufgefaßt werden. Allerdings entsteht diese Variante nicht durch das Einbauen einer Komponente in ein Objekt, wodurch dieses Objekt neue Eigenschaften erhält. Stattdessen wird bei der Definition der Unterklasse festgelegt, welche zusätzlichen Eigenschaften ein Unterklassenobjekt im Vergleich zu einem Oberklassenobjekt besitzt. In Abb. 120 ist die Klasse **Warteschlange** mit ihren drei *Varianten* **Auftragswarteschlange**, **Personenwarteschlange** und **Fahrzeugwarteschlange** dargestellt.

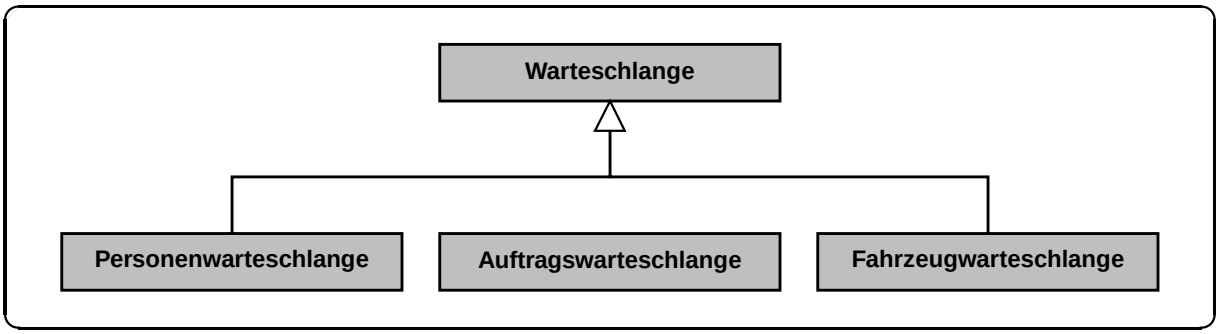


Abb. 120 Varianten der Klasse **Warteschlange**

Eine neue Variante einer Oberklasse kann neue Attribute und Operationen definieren. Außerdem ist es in beschränktem Maße möglich, auf den Kontrollfluß von Oberklassenoperationen Einfluß zu nehmen, sofern eine Oberklassenoperation entsprechend vorbereitet ist. Hierfür läßt sich das *Late-Binding*-Konzept ausnutzen. In dem Beispiel aus Abb. 121 (S. 228) ist die Operation `mA1` der Oberklasse **A** so konstruiert, daß die Unterklassen durch Redefinitionen der Operationen `mA2` und `mA3` Einfluß auf den `mA1`-Kontrollfluß nehmen können. `mA2` und `mA3` stellen damit den von außen *konfigurierbaren* Kontrollfluß dar, während `mInt1` bis `mInt3` den durch **A** festgelegten Kontrollfluß repräsentieren. Natürlich könnte eine Unterklasse `mA1` auch komplett redefinieren.

Grundsätzlich sollte man sich allerdings schon bei dem Warteschlangenbeispiel die Frage stellen, ob aus Modellierungssicht nicht der Einsatz einer Kompositionsbeziehung besser gewesen wäre. Die Klasse **Auftragswarteschlange** entsteht dann unter Verwendung einer Kompositionsbeziehung zu der Klasse **Warteschlange**. In [Tai96] wird darauf hingewiesen, daß die Vererbungsbeziehung auf der Implementierungsebene oft dazu *mißbraucht* wird, eine Kompositionsbeziehung zu simulieren. Neben diesem Aspekt tritt bei der Verwendung der Einfachvererbung zur Variantenbildung das Problem der kombinatorischen Explosion der entstehenden Klassen auf. Liegen beispielsweise drei Komponenten vor, die einem Chassis-Objekt in verschiedenen Kombinationen zugeordnet werden können, so entstehen insgesamt 8 Klassen (vgl. Abb. 122 (S. 228)). Im allgemeinen Fall führt die Verwendung von n Komponenten zu $2^n - 1$ Klassen. Die Anwendung der Einfachvererbung hat zwei weitere Nachteile: Zum einen wird die Funktionalität der Komponenten nur implizit definiert. Die Klasse **ChassisA** enthält die Funktionalität der Komponente **A**, die über das Vererbungskonzept der **Chassis**-Klasse hinzugefügt wird. Eine Weiterverwendung dieser Funktionalität ist nur im Rahmen weiterer Vererbungsbeziehungen möglich, also beispielsweise innerhalb der Klasse **ChassisAB**. In direktem Zusammenhang hiermit steht der zweite Nachteil: Die Komponentenfunktionalität muß innerhalb dieses Entwurfs mehrmals implementiert werden, wie am Beispiel der Operation `mC3` zu erkennen ist, die insgesamt dreimal benötigt wird.

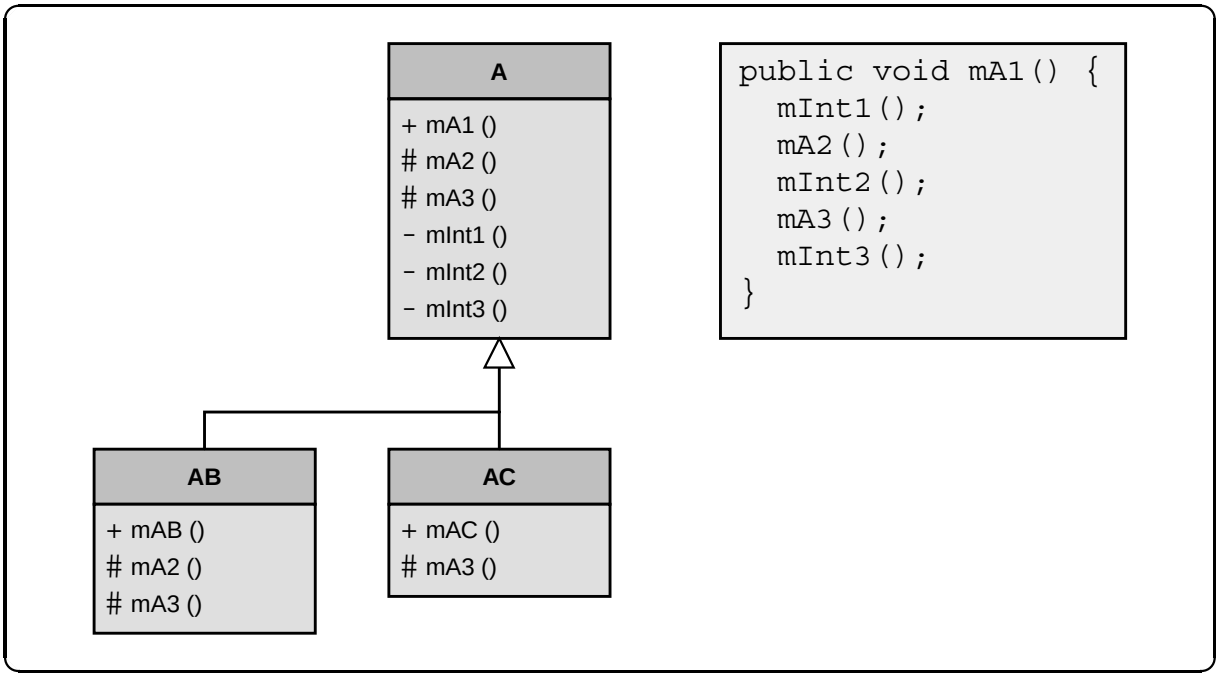


Abb. 121 Änderung des Kontrollflusses in einer Oberklasse

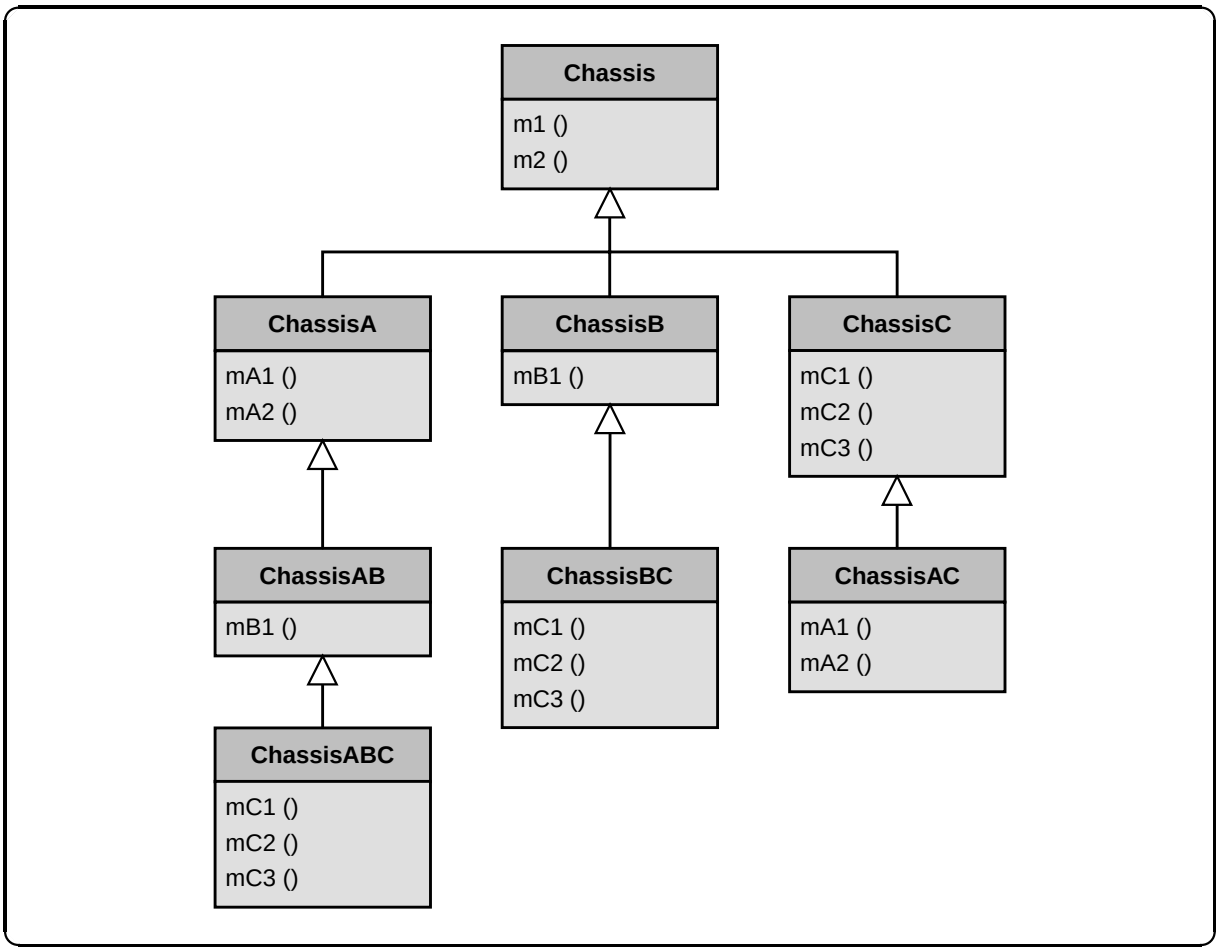


Abb. 122 Variantenbildung mittels Einfachvererbung

Um den Wiederverwendbarkeitsgrad zu erhöhen, kann die Mehrfachvererbung eingesetzt werden. Der in Abb. 123 dargestellte Entwurf beseitigt zwar das Problem der Mehrfachimplementie-

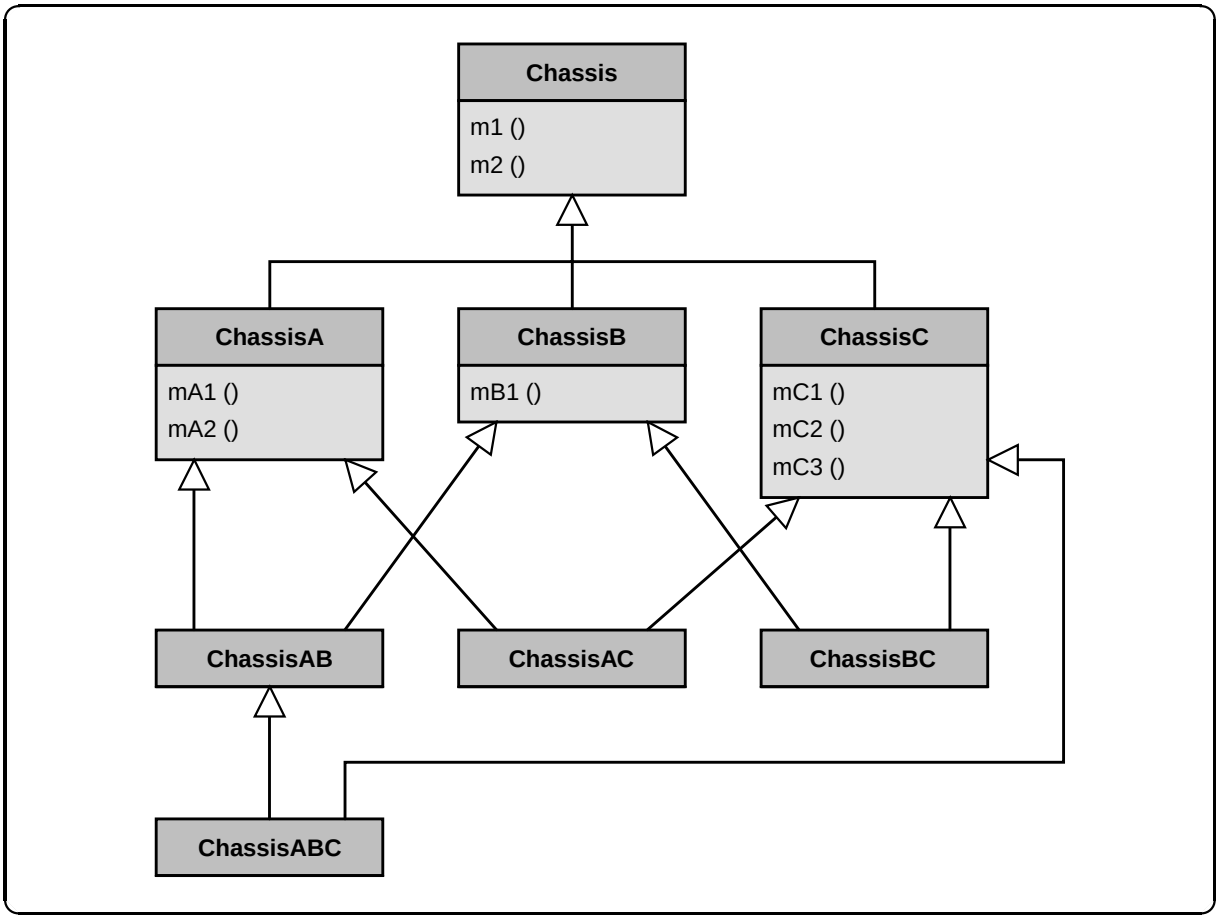


Abb. 123 Variantenbildung mittels Mehrfachvererbung

rung von Methoden, dafür tritt aber das neue Problem der Namenskonflikte auf. Beispielsweise erbt die Klasse **ChassisAB** die Operation `m1` auf zwei Pfaden, einmal über **ChassisA** und zum zweiten über **ChassisB**. Insbesondere wenn in den beiden direkten Oberklassen (z.B.) die Operation `m1` redefiniert wurde, muß innerhalb von **ChassisAB** entschieden werden, wann welche Operation zu verwenden ist. Wählt man dagegen einen Aufbau der Vererbungsstruktur wie in Abb. 124 (S. 230) beschrieben, so besteht das Problem der Namenskonflikte nur noch, wenn in einer der vier Ausgangsklassen **Chassis** und **ComponentA** bis **ComponentC** gleichbenannte Operationen oder Attribute existieren. Der Entwurf aus Abb. 124 (S. 230) benötigt zwar bei der Verwendung von n Komponenten $2^n + n$ Klassen im Vergleich zu den 2^n Klassen des Entwurfs aus Abb. 123, dafür ist die entstehende Struktur deutlich systematischer aufgebaut: eine Klasse **ChassisXYZ** wird *konstruiert*, indem **ChassisXYZ** von der Klasse **Chassis** und den Klassen **ComponentX** bis **ComponentZ** erbt; es gibt also (nach dieser Vorgehensweise) nur einen Weg zur Komposition einer abgeleiteten Chassis-Klasse. In dem Entwurf aus Abb. 123 könnte dagegen die Klasse **ChassisABC** auf mehreren Wegen erzeugt werden: statt von **ChassisAB** und **ChassisC** zu erben, könnte sie z.B. auch von den Klassen **ChassisAC** und **ChassisB** abgeleitet werden.

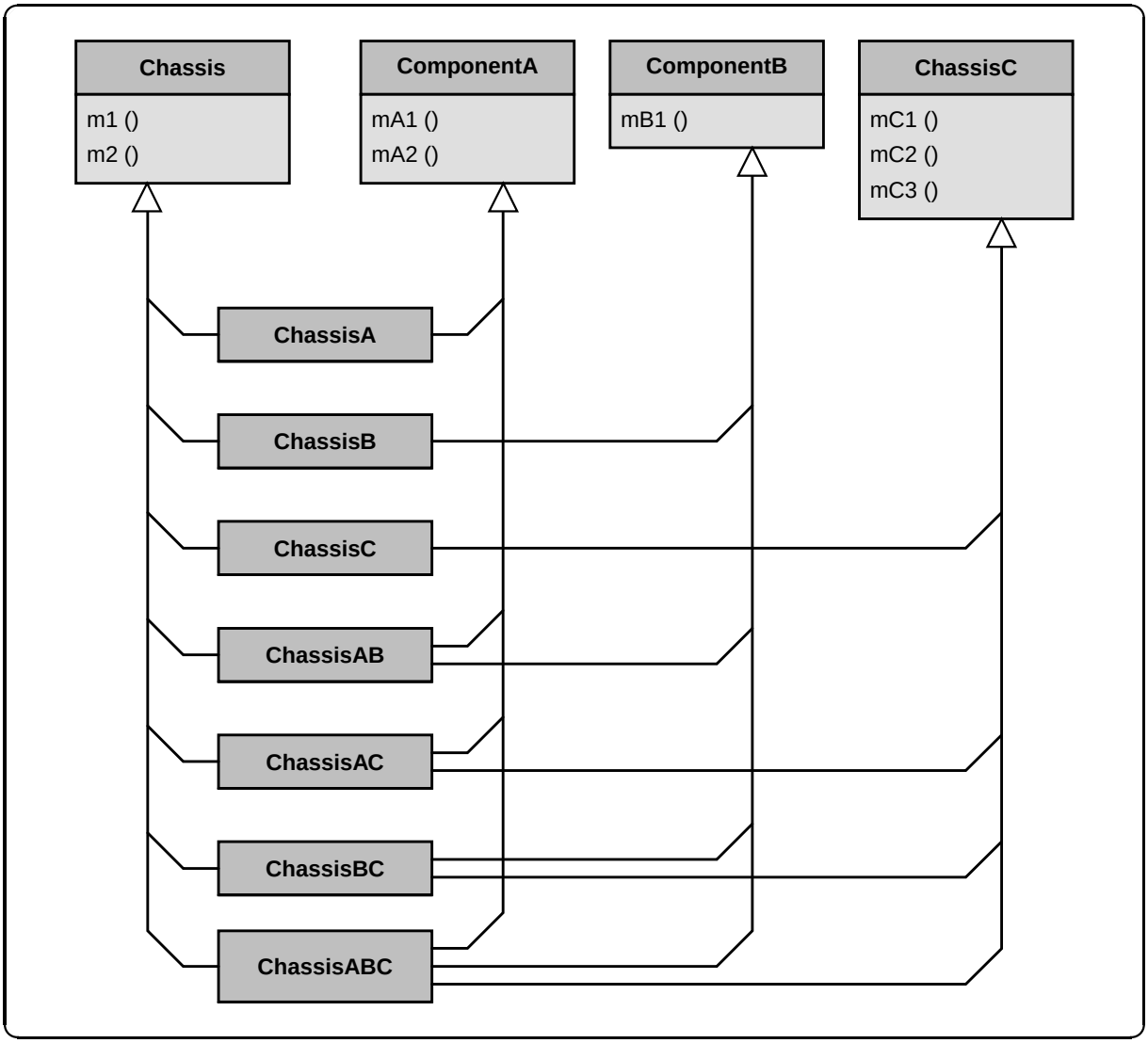


Abb. 124 Variantenbildung mittels Mehrfachvererbung – 2. Ansatz

Betrachtet man die Anforderungen AV1 bis AV6 aus Abschnitt 6.3 (S. 222), so gilt bei der Verwendung des Vererbungskonzepts:

AV1: Durch die Vererbung lassen sich Klassenhierarchien aufbauen. Insofern kann die Forderung nach einer hierarchischen Konstruktion von Bauteilen als erfüllt angesehen werden. Allerdings findet der *Einbau* direkt bei der Objekterzeugung statt, da hier die Klasse des Objekts ausgewählt werden muß. Ein nachträglicher Ein- oder Ausbau von Bauteilen ist damit nicht möglich. Außerdem ist die Vererbung aus konzeptueller Sicht zum Aufbau von *ist-ein*-Beziehungen gedacht, während bei der Variantenbildung die *Whole-Part*-Beziehung im Vordergrund steht.

AV2: Wenn man davon ausgeht, daß die Oberklasse die Chassis-Funktionalität übernimmt, und die Unterklasse die Kombination aus einem Chassis und einer Komponente darstellt, so kann durch die rekursive Anwendung des Vererbungskonzepts die Unterklasse selbst auch wieder als Chassis-Klasse aufgefaßt werden. Bezüglich des Ein- und Ausbaus von Komponenten sowie der konzeptuellen Sichtweise gelten aber wieder die unter

AV1 genannten Einschränkungen.

AV3: Da der Unterklassenanteil eines Objekts, der hier die Komponente repräsentiert, zur Laufzeit nicht aus dem Objekt herausgelöst werden kann, ist die Anforderung AV3 automatisch erfüllt.

AV4: Wenn man pro Chassis-Klasse für eine bestimmte Komponente die Unterklassen explizit neu erstellt, kann man natürlich auch deren Schnittstelle auf die Eigenschaften der Chassis-Klasse abstimmen. Damit läßt sich die Forderung erfüllen, daß die Menge der Operationen einer Komponente von der Chassis-Klasse abhängt. Der Nachteil dieses Ansatzes liegt in der Mehrfachimplementierung der Komponentenfunktionalität. Wählt man dagegen einen der Realisierungsansätze aus Abb. 123 (S. 229) oder Abb. 124 (S. 230), so treten an der Unterklassenschnittstelle immer alle Operationen der Komponente auf. Durch eine Reimplementierung können dann diejenigen Operationen *ausgeblendet* werden, die für diese Chassis-Klasse nicht mehr sichtbar sein sollen. Das *Ausblenden* kann beispielsweise implementiert werden, indem man beim Aufruf einer derartigen Operation eine geeignete Fehlermeldung zurückliefert.

AV5: Geht man wieder davon aus, daß die Chassis-Funktionalität in der Oberklasse und die Komponentenfunktionalität in der Unterklasse liegt, so kann durch Redefinition einer Chassis-Operation in der Unterklasse deren Funktionalität an das Vorhandensein dieser Komponente angepaßt werden. Somit ist AV5 erfüllbar.

AV6: Auch diese Anforderung läßt sich durch die Redefinition von Operationen erfüllen. Bei dem Ansatz aus Abb. 124 (S. 230) könnte beispielsweise in der Klasse **ChassisABC** eine Operation aus **ComponentA** redefiniert werden, um darauf zu reagieren, daß in **ChassisABC** auch die Funktionalität aus **ComponentB** und **ComponentC** vorhanden ist.

7.5 MIXINS und TRAITS

Der letzte Abschnitt hat gezeigt, daß die Mehrfachvererbung für eine Variantenbildung eher geeignet ist als die Einfachvererbung, weil ein höherer Wiederverwendbarkeitsgrad erreichbar ist. Der flexibelste Ansatz hinsichtlich der Wiederverwendbarkeit wird durch den Entwurf aus Abb. 124 (S. 230) erzielt. Die Komponenten werden über eigene Klassen beschrieben und dann zusammen mit der Chassis-Klasse in die abgeleiteten Chassis-Klassen integriert. Da eine Komponentenklasse selbst keine Oberklasse besitzt, kann sie in ihrer internen Struktur keinen Bezug mehr auf die Funktionalität eines Chassis-Objekts nehmen. Typischerweise werden bei einer Komponente aber Annahmen über die Umgebung gemacht, in die sie eingebaut wird: Ein Navigationssystem spezifiziert nicht nur eine physikalische Schnittstelle (z.B. den Aufbau der Steckverbindungen), sondern erwartet auch ein bestimmtes funktionales Verhalten, um selbst funktionsfähig zu sein (z.B. Meßwerte von Sensoren, die für die Standortbestimmung notwendig sind, oder auch die funktionale Schnittstelle zu einem CD-Spieler, der auf Abruf die notwendigen Informationen für eine Routenplanung bereitstellt). Um diese Verbindung zur Umgebung herzustellen, könnte eine Komponentenklasse die Chassis-Klasse erweitern (vgl. den Entwurf aus Abb. 123 (S. 229)). Neben den bereits angesprochenen Namenskonflikten

(vgl. S. 229) tritt dann aber auch das Problem auf, daß man sich auf eine bestimmte Chassis-Klasse festlegen muß, was der Anforderung AV4 (vgl. S. 222) widerspricht.

Eine Möglichkeit, Klassen zu definieren, die sich ausgesprochen flexibel für die Konstruktion neuer Klassen einsetzen lassen, stellt das Konzept der Mixin-Vererbung ([BC90]) dar.

Die Grundidee der Mixin-Vererbung kann an dem folgenden Beispiel aus [ALZ00][4] erklärt werden:

Gegeben seien die JAVA-Klassen `H1` und `P1`.

```
class H1 extends P1 { decs }
```

Dabei stellt `decs` die *Differenz* zwischen den beiden Klassen dar, d.h. die Menge der Attribut- und Methodendeklarationen, die in `H1` im Vergleich zu `P1` hinzukommen. Soll nun eine Klasse `P2` ebenfalls um die `decs`-Deklarationen erweitert werden, so erfordert dieses eine Duplizierung des `decs`-Anteils, was typischerweise durch ein Kopieren ausgeführt werden wird.

```
class H2 extends P2 { decs }
```

Ein besserer Weg besteht darin, `decs` einen eigenen Namen zu geben, und es zu einer Mixin-Klasse zu machen, die dann wie folgt verwendbar ist.

```
mixin M { decs }
class H1 = M extends P1;
class H2 = M extends P2;
```

Wenn sich die `decs`-Deklarationen auf Attribute und Methoden aus `P1` (bzw. `P2`) beziehen, dann stellt `M` keine vollständige Klasse mehr dar. Folglich können von `M` in diesem Fall keine Objekte erzeugt werden. Außerdem wird für die Anwendung einer Mixin-Klasse eine Oberklasse benötigt (im obigen Beispiel `P1` oder `P2`), die mit der Mixin-Klasse kombiniert wird, um insgesamt eine neue Klasse zu erzeugen (im obigen Beispiel `H1` bzw. `H2`). Eine Mixin-Klasse wird in [BC90] daher auch als *abstrakte Unterklasse* bezeichnet.

Das Mixin-Konzept kann jetzt verwendet werden, um eine flexiblere Variantenbildung zu erreichen. Die Komponentenklassen lassen sich als Mixin-Klassen definieren, wobei diese Mixin-Klassen intern Attribute und Operationen verwenden können, die von der Oberklasse, d.h. einer Chassis-Klasse, bereitzustellen sind. Generell kann jede Klasse, die diese Attribute und Operationen zur Verfügung stellt, die Rolle der Oberklasse übernehmen. Im Beispiel aus Abb. 125 (S. 233) werden die beiden Variantenklassen **Chassis_1_AC** und **Chassis_2_ABC** aus den Chassis-Klassen **Chassis_1** und **Chassis_2** sowie den Mixin-Klassen **MixinComponentA** bis **MixinComponentC** erzeugt.

Um das Mixin-Konzept in eine statisch typisierte Programmiersprache zu integrieren, muß eine Mixin-Klasse diejenigen Attribute und Methoden bekanntmachen, die von der Oberklasse erwartet werden, aber selbst in der Mixin-Klasse nicht deklariert sind. In JAM ([ALZ00]) wird hierfür die `inherit`-Deklaration verwendet:

[4] In [ALZ00] wird JAM, eine Erweiterung von JAVA um das Mixin-Konzept, vorgestellt.

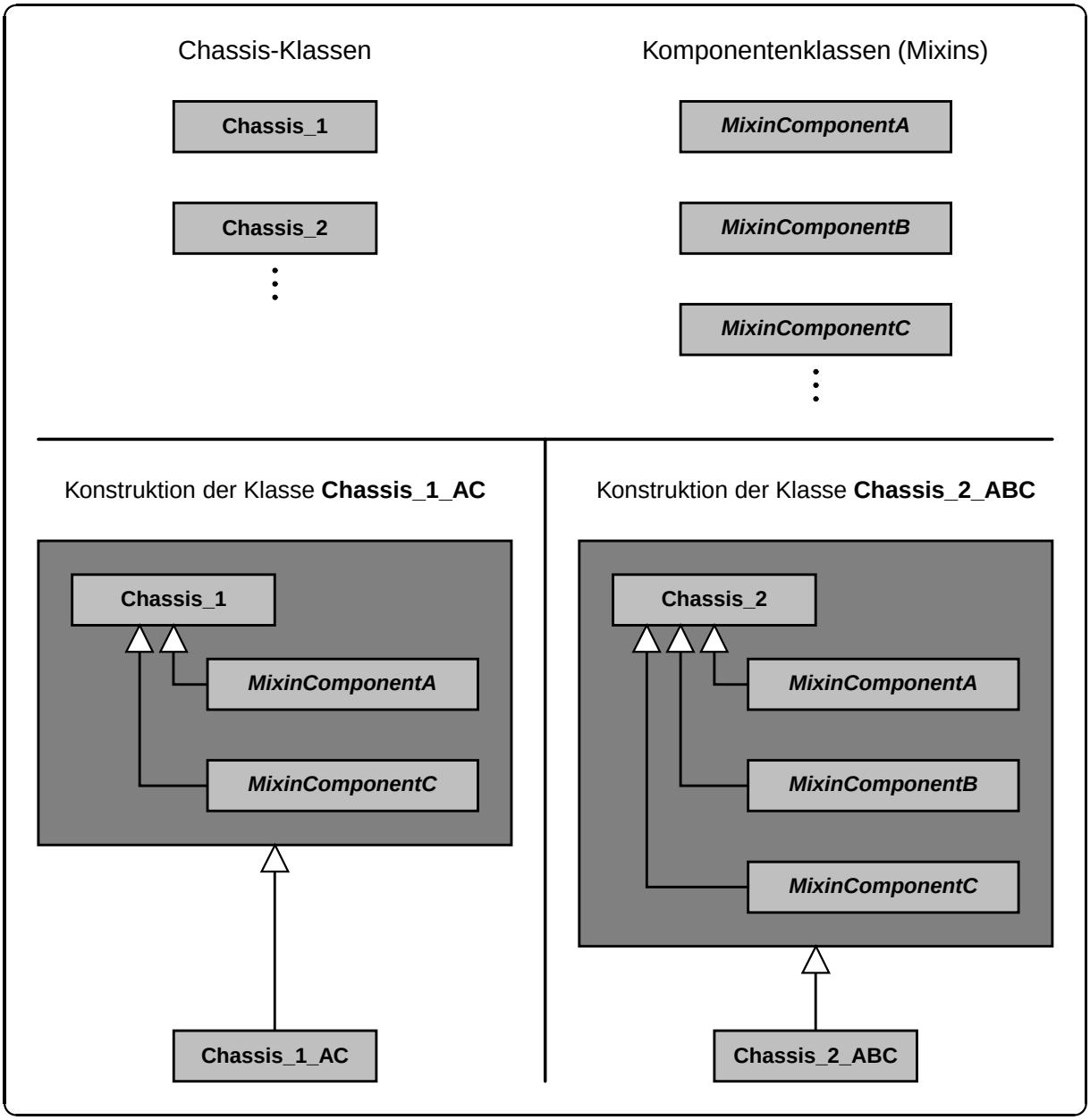


Abb. 125 Variantenbildung durch die Verwendung von Mixin-Klassen

```
mixin ComponentA {
  inherit int x, y;
  inherit void f1 ();
  inherit double f2 (String s);
  lokale Deklarationen
} // ComponentA
```

Eine Oberklasse muß jetzt die beiden `int`-Attribute `x` und `y` sowie die Methoden `f1` und `f2` besitzen, damit sie mit der Mixin-Klasse `ComponentA` kombiniert werden kann.

Das allgemeine Mixin-Konzept enthält die Möglichkeit, mehrere Mixin-Klassen mit einer Oberklasse zu kombinieren.[5] Um die Namenskonflikte aufzulösen, wird bei der Komposition der Mixin-Klassen eine lineare Vererbungsreihenfolge der Mixin-Klassen festgelegt (*Inheritance Chain*). Durch diese Reihenfolge ist dann z.B. vorgegeben, welche Methode aus welcher Mixin-Klasse die anderen, in Konflikt stehenden Methoden, überschreibt.

Einen anderen Ansatz zur Konfliktbehandlung bietet das TRAITS-Konzept an ([SDNB03]). Bevor auf diese Konfliktlösungsstrategie näher eingegangen wird, sollen kurz die Eigenschaften eines Traits vorgestellt werden:

- Ein Trait stellt ein Menge von Methoden zur Verfügung.
- Ein Trait `T` legt fest, welche Methoden eine Klasse `C` oder ein anderer Trait `T1` bereitstellen muß, damit `C` bzw. `T1` den Trait `T` verwenden kann.
- Ein Trait besitzt keine Attribute und damit auch keinen Zustand. Die Methoden eines Traits dürfen auch nicht auf die Attribute der bei der späteren Trait-Verwendung zugeordneten Klasse zugreifen.
- Klassen und Traits können aus anderen Traits zusammengesetzt werden, wobei die Kompositionsreihenfolge unerheblich ist. Auftretende Methodenkonflikte müssen explizit aufgelöst werden.
- Die Verwendung eines Traits in einer Klassendefinition verändert nicht die Semantik der Klasse. Die Kombination aus der Klasse und den Traits hat dieselbe Semantik, wie wenn die Traits-Methoden direkt in der Klasse definiert worden wären. Diese Eigenschaft impliziert, daß eine Trait-Methode nicht eine Methode der Klasse überschreiben darf.
- Die analoge Aussage gilt für das Zusammensetzen eines Traits aus anderen Traits.

In Abb. 126 (S. 235) sind die beiden Traits **TraitA** und **TraitB** sowie die Klassenkomposition mit Traits dargestellt. Die Schnittstelle eines Traits wird durch die Methoden auf der linken Seite beschrieben, während rechts die Methoden stehen, die ein Trait benötigt. Die Beziehung zwischen Klassen und Traits wird in [SDNB03] mit der folgenden Gleichung verdeutlicht:

Class = *Superclass* + *State* + *Traits* + *Glue*

Eine Klasse kann eine Oberklasse besitzen, einen Zustand (in Form von Attributdeklarationen) definieren und einen oder mehrere Traits integrieren. Der *Klebstoff* (*Glue*) wird durch Methoden gebildet, die die Traits verbinden und den Zugriff auf die Attribute ermöglichen. In Abb. 126 (S. 235) stellt die Klasse **ClassC** Methoden für den Zugriff auf die Attribute `x` und `y` zur Verfügung, die selbst wieder von dem Trait **TraitA** benötigt werden. Ein Trait kann nur dann integriert werden, wenn für alle seine benötigten Methoden entsprechende Gegenstücke in der Klasse oder anderen Traits gefunden werden. Das Aneinanderbinden der Methoden findet dabei über die Methodennamen statt.

Für das Auflösen von Namenskonflikten gelten die folgenden Regeln:

- Ist eine Methode sowohl in einer Klasse als auch in einem Trait definiert, so wird die Methode aus der Klasse verwendet.
- Ist eine Methode in einem Trait und in einer Oberklasse vorhanden, so wird die Trait-Methode benutzt. Sie redefiniert dann die Oberklassenmethode.

[5] In der JAM-Realisierung ist dagegen maximal eine Mixin-Klasse erlaubt.

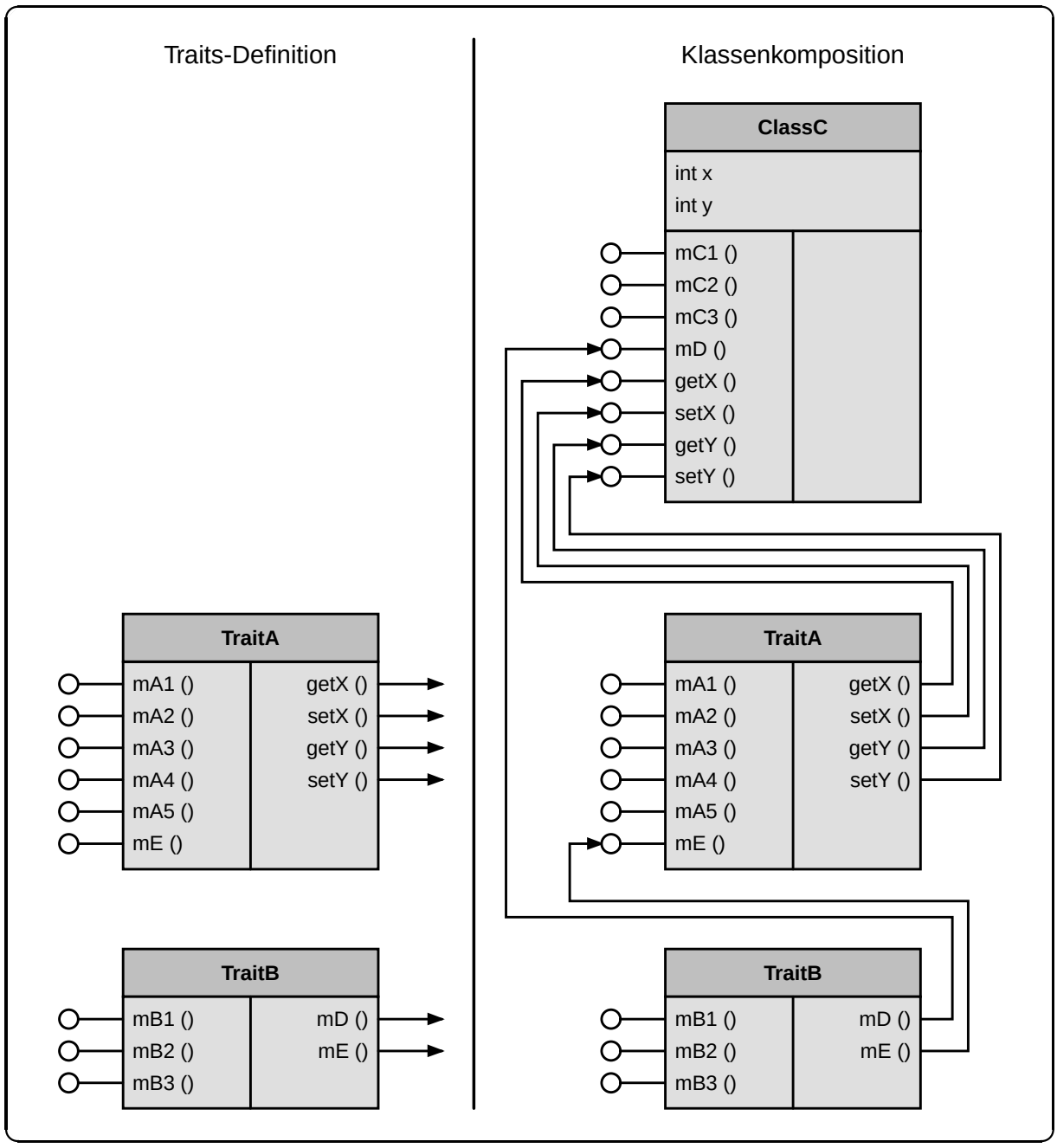


Abb. 126 Variantenbildung durch die Verwendung des TRAITS-Konzepts

- Werden (mindestens) zwei Traits **T1** und **T2** in eine Klasse **C** integriert, die jeweils eine Methode `m` besitzen, dann ist es die Aufgabe der Klasse, diesen Konflikt aufzulösen, indem in dieser Klasse eine neue Methode `m` bereitgestellt wird, die damit die entsprechenden Trait-Methoden redefiniert. Um dabei auf die Trait-Methoden zugreifen zu können, ist es möglich, für diese bei der Traits-Integration *Aliasnamen* zu definieren. Alternativ lassen sich bei der Traits-Integration Methoden, die einen Konflikt erzeugen, explizit ausschließen. Eine Klasse **C** könnte beispielsweise die Methode `m` aus **T1** ausschließen, wodurch automatisch die Methode `m` aus **T2** verwendet wird.
- Dieselben Regeln gelten, wenn zwei Traits **T1** und **T2** in einen Trait **T3** integriert werden.

Sowohl das Mixin- als auch das Traits-Konzept stellen die Bindung zwischen den einzelnen syntaktischen Einheiten über die Namensgleichheit her. Ein Trait läßt sich beispielsweise nur dann von einer Klasse verwenden, wenn diese entsprechend benannte Methoden anbietet, die von dem Trait benötigt werden. In [Ern02] wird das Mixin-Konzept dahingehend weiterentwickelt, daß eine Mixin-Klasse formale Parameter spezifizieren darf, wobei auch Methodenparameter erlaubt sind. Durch diese Parametrisierung können die Mixin-Klassen mit jeder Klasse kombiniert werden, die Attribute der entsprechenden Typen bzw. Methoden mit den geforderten Signaturen besitzt. Das folgende Beispiel zeigt die Verwendung einer Mixin-Klasse `ColorMixin`, um zusammen mit der Klasse `Point` die Klasse `ColorPoint` zu konstruieren:

```
class Point {
  int x,y;
  void print() {
    System.out.println ("("+x+","+y+")");
  } // print
} // Point

mixin ColorMixin (int a; int b; void write();) {
  String color;
  void write () {
    System.out.println ("("+a+","+b+","+color+")");
  } // write
} // ColorMixin

class ColorPoint = Point (x,y,print) ColorMixin;
```

Im Vergleich zu dem *konventionellen* Vererbungskonzept aus Abschnitt 7.4 (S. 226) sind Mixins und Traits Bausteine, die sich flexibler einsetzen lassen. Insbesondere Traits führen durch die Entschärfung der Namenskonfliktproblematik zu insgesamt stabileren Software-Strukturen. Bezogen auf die Anforderungen AV1 bis AV6 aus Abschnitt 6.3 (S. 222) gelten jedoch dieselben Aussagen wie für das Vererbungskonzept (vgl. S. 230).

7.6 Variational Object-Oriented Programming: Das RONDO-Modell

Das RONDO-Modell ([Mez98], [Mez96]) verfolgt als wesentliches Ziel, eine inkrementelle Spezifikation von Verhaltensvariationen (*Incremental Behavior Variation*) zu unterstützen. Die zugrunde liegende Philosophie besteht in einer konsequenten Trennung zwischen dem Basisverhalten eines Objekts und kontextabhängigen Variationen dieses Basisverhaltens. In [Mez98, Abschnitt 2.2] werden vier kontextabhängige Verhaltenstypen identifiziert:

- **Zustandsabhängiges Verhalten** (*State-Dependent Behavior*): Die Semantik einer Operation hängt vom Objektzustand ab. Ein Beispiel ist das Abbuchen von einem Girokonto. Wenn der Überziehungskredit ausgeschöpft ist, sind weitere Abbuchungen nicht mehr erlaubt.

- **Von der Betrachtungsperspektive abhängiges Verhalten** (*Perspective-Dependent Behavior*): Für ein Objekt können bezüglich seiner Funktionalität in Abhängigkeit des aktuellen Betrachters unterschiedliche Sichtweisen existieren. Ein typisches Beispiel sind Personen, die unterschiedliche Funktionen übernehmen, wodurch sich die Semantik einer Operation, je nach der gerade verwendeten Sichtweise, verändern kann.
- **Anwendungsabhängiges Verhalten** (*Application-Dependent Behavior*): Für ein Objekt liegen mehrere Implementierungen vor, und die das Objekt nutzende Applikation ist in der Lage, im Rahmen eines Adaptionsprozesses eine geeignete Implementierung auszuwählen. Eine statische Adaption liegt vor, wenn zum Zeitpunkt der Objekterzeugung die Implementierung ausgewählt wird, und dann zur Laufzeit nicht mehr änderbar ist. Bei einer dynamischen Adaption kann (zusätzlich) auch zur Laufzeit die Implementierung ausgetauscht werden. Eine wichtige Anwendungsklasse ist die Anpassung von Strategien im Betriebssystembereich. Viele Anwendungen werden bei der Speicherverwaltung sehr gut durch eine *Least-Recently-Used*-Strategie (vgl. [NS98, S. 82]) unterstützt. Liegt allerdings eine Datenbankanwendung vor, die sequentiell einen großen Datenbestand verarbeitet, so wird eine *First-In-First-Out*-Strategie (vgl. [NS98, S. 82]) besser geeignet sein.
- **Umgebungsabhängiges Verhalten** (*Environment-Dependent Behavior*): Auch hier liegen wieder mehrere Objektimplementierungen vor. Die Auswahl der geeigneten Implementierung ist von der aktuellen Ausführungsumgebung abhängig, z.B. von dem vorhandenen Speicherplatz oder der Prozessorleistung.

Das Basisverhalten eines Objekts wird durch seine Klassenzugehörigkeit bestimmt. Kontextspezifische Verhaltensänderungen werden im RONDO-Modell über **Adjustments** spezifiziert, die sich dann objektbezogen aktivieren und deaktivieren lassen.

Eine Adjustment-Deklaration wird einer Klasse oder einer anderen Adjustment-Deklaration zugeordnet und besitzt die folgende Struktur ([Mez98, S. 86]):

- Es dürfen neue Attribute (Variablen) definiert werden.
- Es dürfen neue Methoden definiert werden.
- Methoden der später zugeordneten Klasse bzw. des zugeordneten Adjustments können redefiniert werden. Dabei ist es möglich, mit `super` auf die ursprüngliche Funktionalität zuzugreifen.

Für die Adjustment-Zuordnung existieren zwei prinzipielle Wege:

- Die Verwendung der `modifies`-Klausel.
- Die Verwendung der `connects`-Klausel.

Beide Klauseln können um einen optionalen `when`-Teil ergänzt werden. Der Aufbau und die Bedeutung des `when`-Teils wird zunächst am Beispiel der `modifies`-Klausel erklärt und kann dann analog auf die `connects`-Klausel übertragen werden. Für die syntaktische Struktur der `modifies`-Klausel gibt es vier Varianten, wobei A einen Adjustment-Namen und M einen anderen Adjustment-Namen oder einen Klassennamen darstellt.

- M1: A `modifies:` M
  A stellt eine *Default*-Variation des Verhaltens von M dar und wird automatisch aktiviert, wenn ein Objekt der zugehörigen Klasse erzeugt wird. Ist M ein Adjustment-Name, so

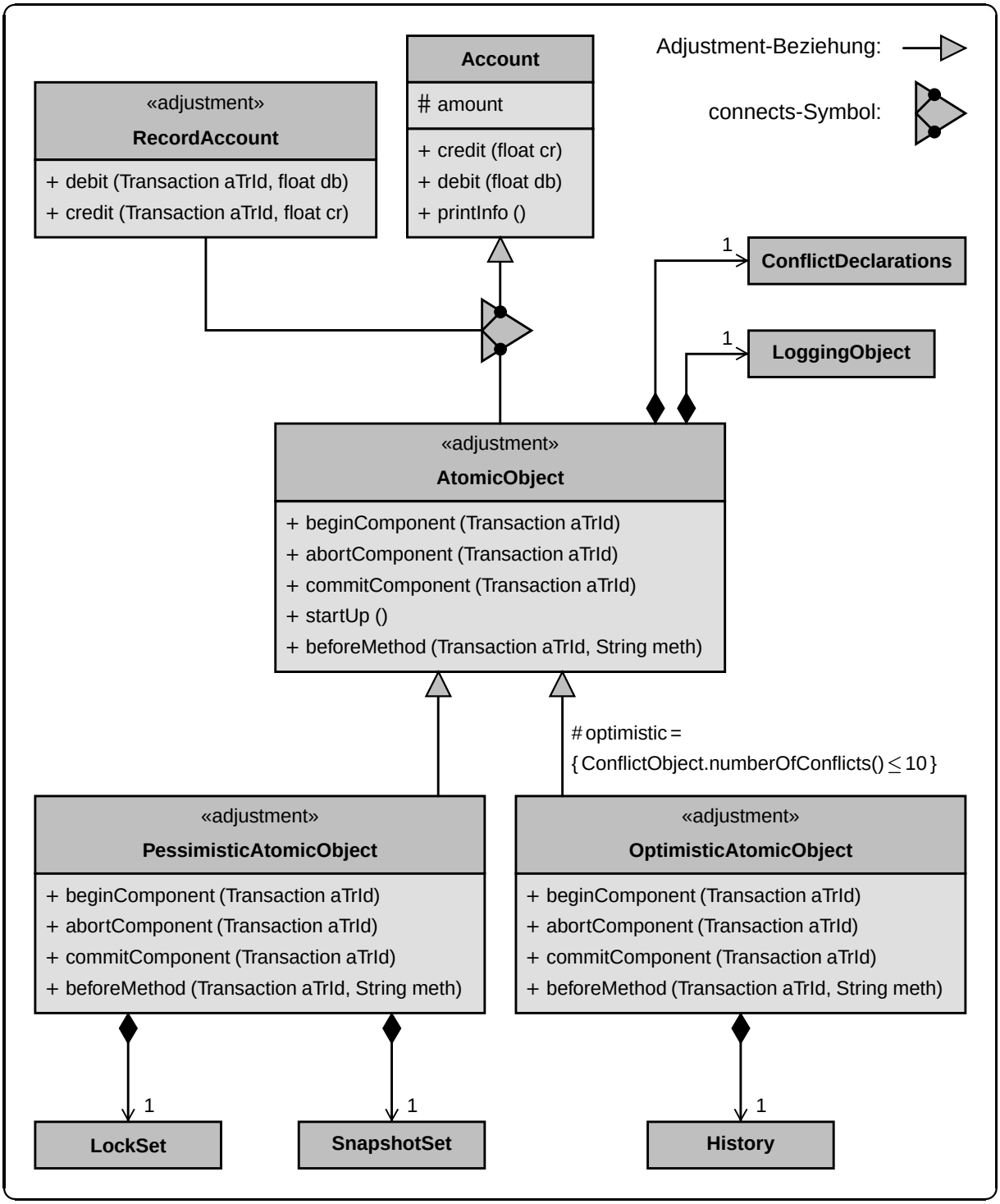


Abb. 127 Das RONDO-Modell für den Ausschnitt einer Bankanwendung – Teil 1

wird A aktiviert, wenn M aktiviert wird. In Abb. 127[6] stellt `PessimisticAtomicObject` eine Default-Adjustment für die Adjustment `AtomicObject` dar, d.h. es wird zunächst ein pessimistisches Synchronisationsverfahren verwendet, z.B. unter Anwendung eines Sperrkonzepts.

[6] Das Beispiel ist [Mez98, S. 91 ff] entnommen und wird hier in einer erweiterten UML-Notation dargestellt, um die Beziehungen zwischen den einzelnen Klassen und Adjustments stärker hervorzuheben.

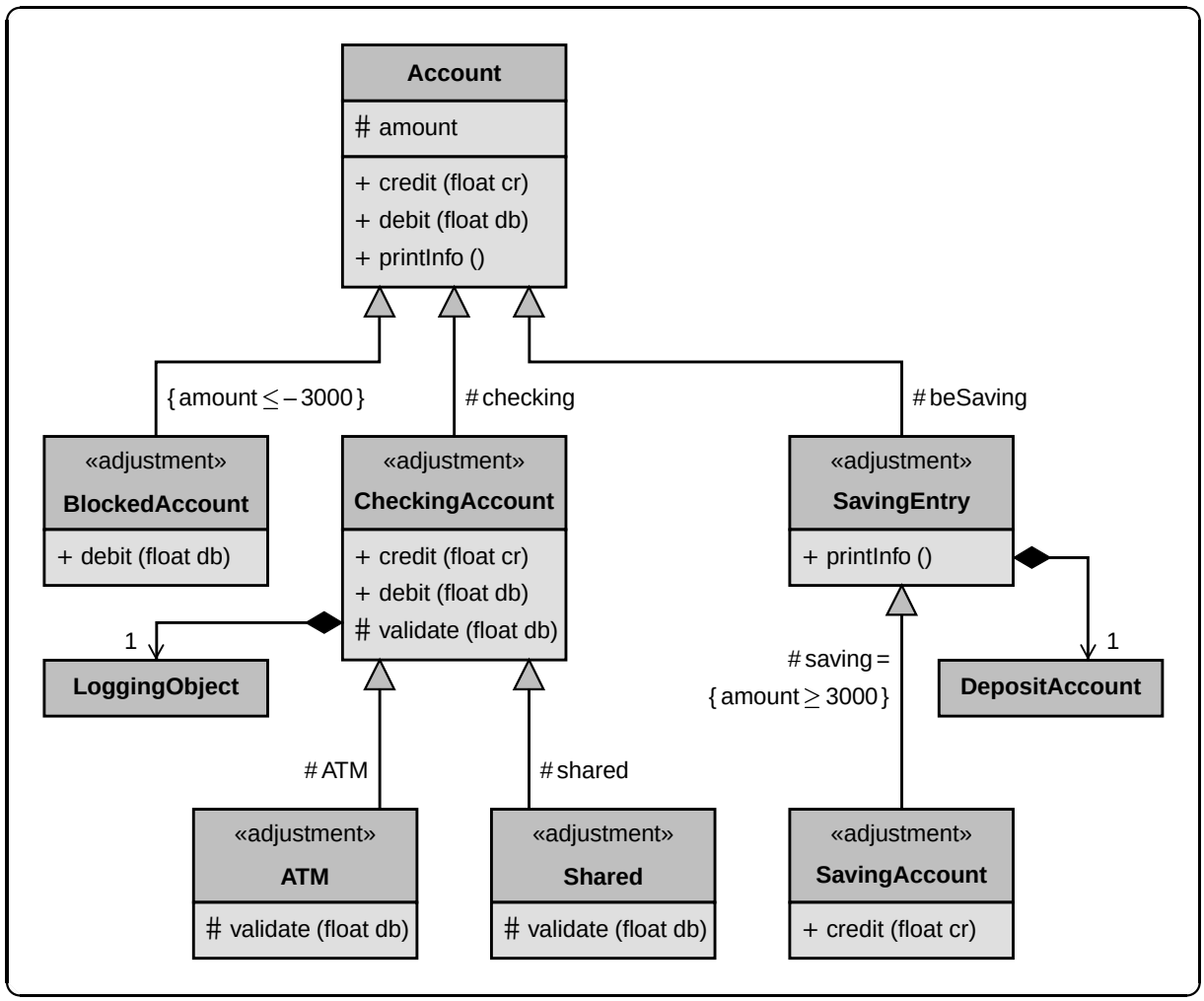


Abb. 128 Das RONDO-Modell für den Ausschnitt einer Bankanwendung – Teil 2

- M2: *A* `modifies:` *M* `when:` *`#eventName`*
  Für die Aktivierung von *A* gibt es zwei Wege: (1) *`#eventName`* kann bei der Objekterzeugung angegeben werden. (2) Unter der Annahme, daß `m` eine Referenz auf ein *M*-Objekt darstellt, kann durch den Aufruf von `m.raise(`*`#eventName`*`)` *A* aktiviert werden. Durch den Aufruf von `m.undo(`*`#eventName`*`)` wird *A* wieder entfernt. In Abb. 128 könnte beispielsweise das Bankkonto `acc` durch den Aufruf von `acc.raise(#checking)` in ein *überwachtes* Konto abgeändert werden. Damit wird jetzt die redefinierte `debit`-Methode aus `CheckingAccount` verwendet, die unter Anwendung der `validate`-Methode prüft, ob die Auszahlung erlaubt ist.
- M3: *A* `modifies:` *M* `when:` {*`condition`*}
  Ist die Bedingung erfüllt, dann wird *A* aktiviert, andernfalls deaktiviert. Die Auswertung der Bedingung findet jedesmal statt, wenn eine Methode aus *A* aufgerufen wird. Für das Bankkonto aus Abb. 128 gilt: Liegt ein Kontostand ≤ -3000 vor, so wird `BlockedAccount` aktiviert. Damit ist ein weiteres Geldabheben nicht mehr möglich, weil die Redefinition der `debit`-Methode dieses verhindert.[7]

[7] Die spezifizierte Bedingung verhindert allerdings nicht, daß ein `debit`-Aufruf das Konto beliebig überzieht, da bei einem Kontostand von –2900 noch ein beliebig hoher Betrag abgehoben werden kann.

- M4: *A* `modifies:` *M* `when:` *#EventName* = {*condition*}
  Liegt die Kombination aus einem Ereignisnamen und einer Bedingung vor, so steht (automatisch) eine Methode `switch`*EventName*`Context` zur Verfügung. Der Aufruf dieser Methode testet, ob die Bedingung erfüllt ist; falls ja, wird ein `raise(`*EventName*`)` aufgerufen, andernfalls ein `undo(`*EventName*`)`.
  Durch den Aufruf `acc.switchSavingContext()` für ein Konto `acc` wird in dem Beispiel aus Abb. 128 (S. 239) `SavingAccount` aktiviert, falls der aktuelle Kontostand einen Wert $\geq$ 3000 besitzt. Andernfalls wird `SavingAccount` deaktiviert.[8]

Die `connects`-Klausel besitzt die folgende Struktur:

A `connects:` {*M1*,*M2*} *when-Klausel*

M1 und *M2* sind Adjustment- oder Klassennamen. Ist *A* aktiviert, so definieren deren Methoden, wie das Zusammenspiel der Methoden von *M1* und *M2* sein soll. In Abb. 127 (S. 238) stellt `RecordAccount` eine Verbindung zwischen `Account` und `AtomicObject` her. Da keine `when`-Klausel angegeben wurde, ist einem Konto immer die `AtomicObject`-Funktionalität zugeordnet.

RONDO wurde (prototypisch) durch eine explizite Erweiterung der Metaebene von SMALLTALK-80 realisiert ([Mez98, S. 189]).

Das RONDO-Modell erlaubt über das Adjustment-Konzept eine sehr flexible Objektspezialisierung. Bei der Variantenbildung für die Objekte liegt aber, wie bei den Vererbungskonzepten, der Schwerpunkt auf der *ist-ein*-Beziehung. Änderungen des Kontrollflusses, die sich durch die Aktivierung oder Deaktivierung von Adjustments ergeben, werden über die Methodenredefinition spezifiziert. Da RONDO die Aktivierung und Deaktivierung von Adjustments zur Laufzeit zuläßt, können die Anforderung AV1 und AV2 (vgl. S. 222) einfacher erfüllt werden als beim Vererbungskonzept, allerdings liegt eine explizite Unterstützung der Kompositionsbeziehungen nicht vor. Auch AV4 wird unterstützt, da durch die Aktivierung eines Adjustments neue Methoden an der Objektschnittstelle sichtbar werden. Insgesamt betrachtet steht aber auch bei diesem Ansatz wieder deutlich der Vererbungscharakter im Vordergrund.

7.7 Hyperspaces und HYPER/J

In [TOHS99] und [OT01a] wird ein Ansatz vorgestellt, der eine Weiterentwicklung der Subjektorientierten Programmierung ([HO93]) darstellt und als *Multi-Dimensional Separation of Concerns* bezeichnet wird. Als ein Hauptproblem für die Evolution, Integration und Wiederverwendung von Software-Komponenten wird die folgende Eigenschaft heutiger Software-Architekturen identifiziert: Obwohl beim Entwicklungsprozeß das Konzept der Trennung von Zuständigkeiten (*Separation of Concerns*, [Par72]) eine wesentliche Rolle spielen sollte, wird der Systementwicklungsbereich tatsächlich von *einem* Zerlegungskriterium dominiert, was in [TOHS99] als *Tyranny of the Dominant Decomposition* charakterisiert wird. Nimmt man die Objektorientierung als Grundlage, so stellt die Klasse die zentrale Zerlegungseinheit dar. Damit wird der Datenaspekt in den Vordergrund gestellt, da eine Klasse zusammengehörende Daten (z.B. die

[8] Ein Aufruf von `switchSavingAccount()` ist natürlich nur erlaubt, wenn zuvor `SavingEntry` aktiviert wurde.

Daten eines Mitarbeiters) kapselt. Soll eine neue Funktionalität, beispielsweise die Visualisierung an der Benutzungsschnittstelle, in ein bestehendes System integriert werden, so sind in der Regel sehr viele Klassen davon betroffen.

Um bei der Systementwicklung alle Zerlegungskriterien berücksichtigen zu können, wird das Konzept der **Hyperspaces** vorgeschlagen. Ein Hyperspace ist ein n-dimensionaler Raum, wobei jede **Dimension** ein Zerlegungskriterium repräsentiert. Abb. 129 zeigt ein Beispiel für einen Hyperspace, das [OT01b] entnommen wurde.[9]

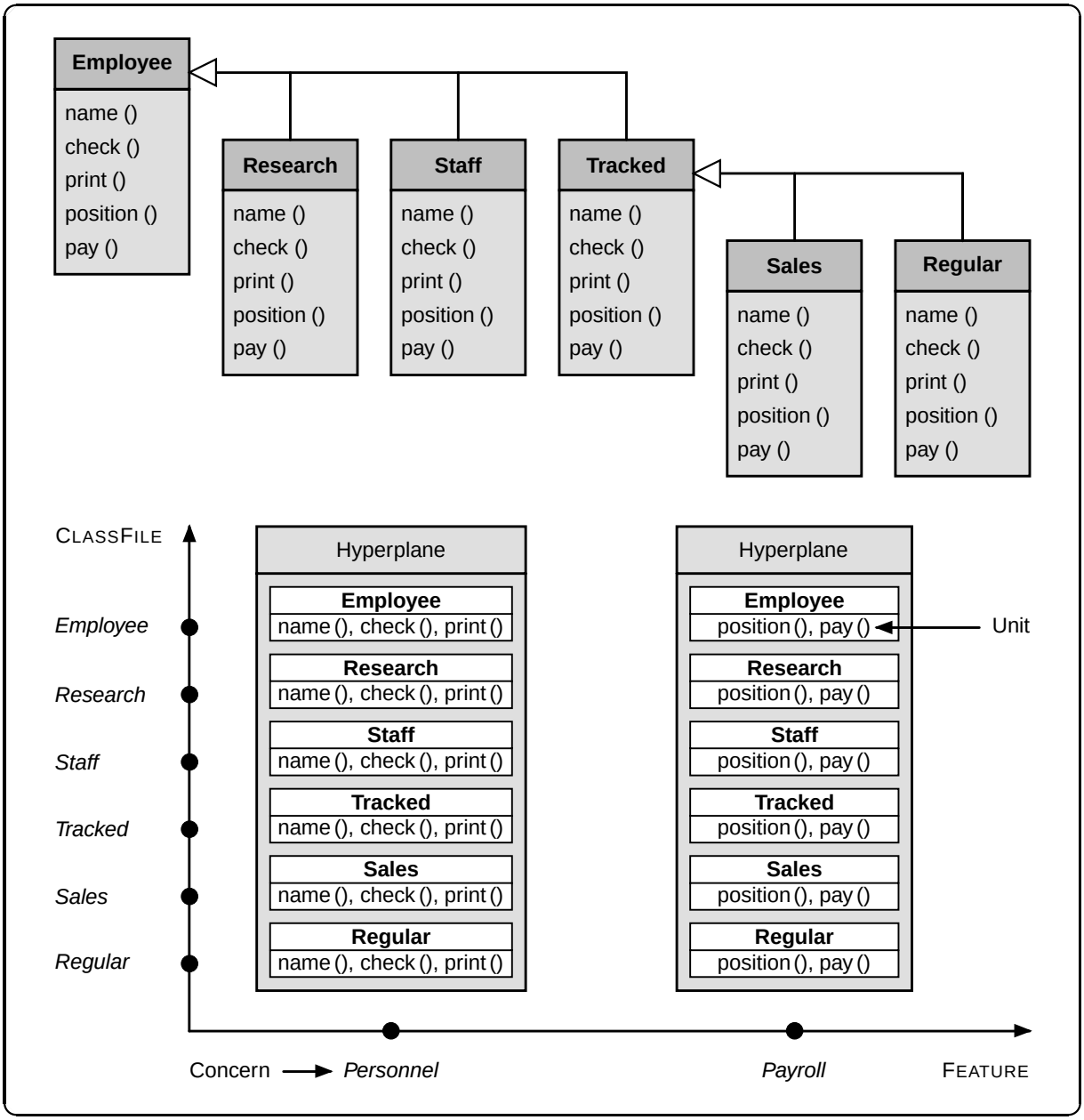


Abb. 129 Definition eines Hyperspaces

[9] Um das Beispiel übersichtlich zu gestalten, wurden einige Unterklassen weggelassen.

Die CLASSFILE-Dimension ist standardmäßig vorhanden, wenn HYPER/J für die Definition eines Hyperspaces verwendet wird. HYPER/J (vgl. [TO00]) ist ein Werkzeug, das zur Dekomposition und Kompostion von JAVA-Anwendungen verwendbar ist.[10] HYPER/J arbeitet auf JAVA class-Dateien. Der JAVA-Quelltext der Klassen wird nicht benötigt.

Die FEATURE-Dimension wurde definiert, um die Klassen bezüglich des Personal- und des Lohnabrechnungsaspekts zerlegen zu können, die jeweils die Rolle eines **Concerns** spielen. Für jede Dimension lassen sich (beliebig viele) Concerns definieren, wobei die CLASSFILE-Dimension automatisch die im Hyperspace vorhandenen JAVA-Klassen als Concerns enthält.

Für einen Hyperspace werden zusätzlich Grundeinheiten (**Unit**) definiert. Bei HYPER/J sind dies die Attribute (Instanzvariablen) und Methoden der Klassen. In jeder Dimension müssen alle Units des Hyperspaces den Concerns zugeordnet werden. Der in Abb. 129 (S. 241) definierte Hyperspace enthält insgesamt 30 Units, die jeweils 5 Methoden der 6 Klassen. Die Methode `check` der Klasse **Staff** ist in der CLASSFILE-Dimension dem Concern *Staff* und in der FEATURE-Dimension dem Concern *Personnel* zugeordnet.

Alle Units, die zu einem Concern gehören, definieren eine **Hyperplane**. In dem Hyperspace aus Abb. 129 (S. 241) existieren somit 8 Hyperplanes.

Für die Dekomposition eines Systems sind damit die folgenden Schritte durchzuführen:

- Definition eines Hyperspaces.
- Auswahl der Klassen und Schnittstellen, die zu dem Hyperspace gehören sollen (z.B. alle Klassen eines Pakets oder Teilmengen eines Pakets).
- Definition der Dimensionen des Hyperspaces.
- Definition der Concerns der einzelnen Dimensionen.
- Abbildung (Zuordnung) der Units (Attribute, Methoden) aller Hyperspace-Klassen auf die Concerns der verschiedenen Dimensionen.

Die Methodenmenge einer Hyperplane muß nicht notwendigerweise abgeschlossen sein. In der Regel wird es Methoden geben, die intern Methoden aus anderen Hyperplanes verwenden. Um in sich abgeschlossene Modellierungseinheiten zu erhalten, werden **Hyperslices** definiert. In HYPER/J entsteht aus einer Hyperplane eine Hyperslice, indem für die fehlenden Methoden automatisch abstrakte Methoden zur Verfügung gestellt werden. Die Hyperslices bilden die Grundlage für den **Kompositionsprozeß**. Neue Systeme oder Systemteile können jetzt über **Hypermodule** definiert werden:

- Einem Hypermodul werden eine oder mehrere Hyperslices zugeordnet.
- Durch die Spezifikation von *Kompositionsregeln* (**Composition Rule**) wird festgelegt, welche Units der einzelnen Hyperslices den neu zu erzeugenden Klassen zugeordnet werden. Wichtige Beispiele für Kompositionsregeln sind:[11]
  - **mergeByName**: Alle Units, die in den angegebenen Hyperslices demselben Klassennamen zugeordnet sind, werden in die neue, zusammengesetzte Klasse übernommen.
  - **merge**: Es werden Units (typischerweise Klassen) zusammengesetzt, deren Namen nicht identisch sind.

[10] HYPER/J kann von `http://www.research.ibm.com/hyperspace` bezogen werden.

[11] Eine vollständige Beschreibung der HYPER/J-Kompositionsregeln findet sich in [TO00].

- **equate**: Hiermit ist es z.B. möglich, einer abstrakten Operation einer Hyperslice eine oder mehrere konkrete Operationen zuzuordnen. Werden mehrere Operationen angegeben, so werden diese hintereinander ausgeführt.
- **bracket**: Mit `bracket` ist es möglich, einer Menge von Methoden `before`- und `after`-Methoden zuzuordnen.

Da ein Hypermodul selbst wieder eine Hyperslice ist, können Hypermodule Bestandteile für die Definition neuer Hypermodule sein.

Das Hyperspace-Konzept läßt sich in den Systementwicklungsprozeß auf verschiedene Arten integrieren:

- Neukonzeption eines Systems:
  Die einzelnen Zuständigkeitsbereiche (Concerns) werden zunächst unabhängig voneinander entwickelt. Durch die Definition eines Hypermoduls findet dann die Systemkomposition statt.
- Anpassung eines bestehenden Systems:
  Innerhalb der Definition eines neuen Hypermoduls für das Gesamtsystem lassen sich gezielt ganze Pakete, Klassen oder auch nur einzelne Methoden ersetzen.
- Erzeugung von Systemvarianten mit reduzierter Funktionalität:
  Soll das System in einem Anwendungsbereich eingesetzt werden, in dem eine bestimmte Funktionalität nicht benötigt wird, so kann durch eine Dekomposition diese Funktionalität identifiziert werden. In der folgenden Systemkomposition über ein neues Hypermodul bleibt sie dann unberücksichtigt.
- Erweiterung der Systemfunktionalität:
  Es wird ein neues Hypemodul definiert, das neben der alten Systemfunktionalität auch die Hyperslices (und damit Klassen) enthält, die die neue Systemfunktionalität zur Verfügung stellen. Durch die Verwendung entsprechender Kompositionsregeln wird diese dann in das alte System integriert, z.B. über den Mechanismus der before- und after-Methoden.

Insgesamt ist es über die *On-Demand Remodularization* Funktionalität von HYPER/J möglich, neue statische Systemvarianten zu erzeugen. Die Grundlage für die Variantenbildung stellen aber Klassen und nicht Objekte dar. Eine Variantenbildung von Objekten durch einen Ein- und Ausbau von Bauteilen wird explizit nicht unterstützt. Auch sind bei der Variantenbildung nur sehr eingeschränkte Anpassungen des Kontrollflusses möglich: beispielsweise kann festgelegt werden, (unterschiedliche) Operationen gleichen Namens zu kombinieren, indem sie hintereinander ausgeführt werden. Eine direkte Unterstützung der Anforderungen AV1 bis AV6 (vgl. S. 222) ist daher nicht vorhanden.

7.8 Adaptive Programmierung: Die DEMETER-Methode und DJ

Die adaptive Programmierung verfolgt das Ziel, die enge Kopplung zwischen der Ausführung von Algorithmen und der Objektstruktur aufzuheben. Soll beispielsweise die Gehaltssumme einer Firma berechnet werden, so geht die Gesamtstruktur eines Firmenobjekts in die Berechnung ein. Wird als Grundlage das linke Klassendiagramm aus Abb. 130 (S. 244) verwendet, so könnte die Methode `berechneGehaltssumme` wie folgt aussehen:

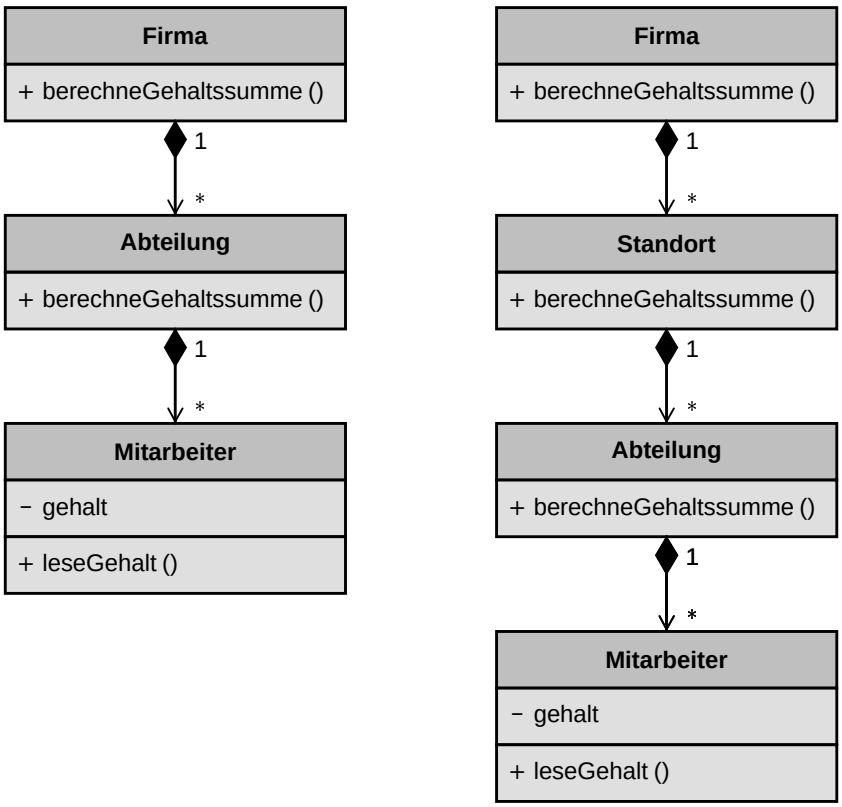


Abb. 130 Die Struktur einer Firma (vgl. [CE00, S. 261])

```
public double berechneGehaltssumme () {
  double summe = 0.0;
  for ( int i=0 ; i<abteilungen.size() ; i++ ) {
      Abteilung abt = (Abteilung) abteilungen.elementAt(i);
      summe += abt.berechneGehaltssumme ();
  } // for
  return summe;
} // berechneGehaltssumme
```

Wird die Firma auf mehrere Standorte verteilt (vgl. rechtes Klassendiagramm in Abb. 130), so muß die Methode `berechneGehaltssumme` der Klasse Firma geändert werden, um die neue Firmenstruktur zu berücksichtigen, obwohl die Berechnungsvorschrift eigentlich gleich bleibt: es sind die Gehaltssummen der untergeordneten Verwaltungseinheiten zu berechnen und aufzuaddieren.

Die DEMETER-Methode ([Lie96]) ist ein Verfahren zur adaptiven Programmierung. Ein DEMETER-Programm besteht aus einer Struktur- und einer Verhaltensbeschreibung:

- Die **Strukturbeschreibung** findet über ein **Klassen-Dictionary** statt. Eine Klassenstruktur wird hierbei in Form einer Graphrepräsentation beschrieben. Das linke Klassendiagramm aus Abb. 130 wäre ein Graph mit drei Knoten, während das rechte Klassendiagramm einen Graphen mit vier Knoten darstellt.

- Für die **Verhaltensmodellierung** werden **Propagation Patterns** benutzt, deren Beschreibung aus drei zentralen Teilen besteht:
  - Die **Signatur** der zugeordneten Methode.
  - Die **Propagation-Direktive** (in späteren DEMETER-Versionen auch als **Traversal Strategy** bezeichnet), die in Form einer generischen Spezifikation die betroffenen Klassen festlegt.
  - Eine **Wrapper**-Definition, die festlegt, welche algorithmische Funktionalität die Methode ausführen soll.

Für das Beispiel der Gehaltssummenberechnung wäre die generische Propagation-Direktive `from Firma to Mitarbeiter` sinnvoll, da die Gehaltssumme, unabhängig von der Organisationsstruktur der Firma, von den individuellen Mitarbeitergehältern abhängt. Die Propagation-Direktive läßt sich jetzt auf alle Klassengraphen anwenden, die einen **Firma**- und einen **Mitarbeiter**-Knoten besitzen. Um ein konkretes Programm zu erzeugen, wird das Propagation-Pattern mit einem Klassengraphen kombiniert. Es wird ein Teilgraph erzeugt, der alle Knoten enthält, die auf den Pfaden vom Startknoten zu den Endknoten[12] liegen. Für jede Klasse `K` des Teilgraphen wird eine Methode mit der angegebenen Signatur generiert, die neben der Funktionalität aus der Wrapper-Definition den *Traversal-Code* enthält, der sich aus der Struktur des Graphen ergibt. Dieser Code besteht im wesentlichen aus Methodenaufrufen an die Objekte, die zu den Klassen gehören, welche direkte Nachfolgerknoten von `K` sind.

Die aktuellen Versionen der DEMETER-Werkzeuge zur Programmgenerierung für die Sprachen C^{++} und JAVA (DEMETERJ) sind unter [Dem03] erhältlich. In [OL01] wird DJ, eine dynamische Variante von DEMETERJ, vorgestellt. Die Propagation Strategy kann hier direkt innerhalb eines JAVA-Programms definiert werden, wobei als Strukturbeschreibung des Klassengraphen ein Objekt der Klasse `ClassGraph` dient. Dieses Objekt repräsentiert alle Klassen eines Pakets. Unter Verwendung des JAVA Reflection APIs wird der Klassengraph aufgebaut. Die Wrapper-Funktionalität wird über ein Objekt der Klasse `Visitor` bereitgestellt. Damit ist eine statische Generierung wie bei DEMETERJ nicht mehr notwendig, was flexiblere Programmstrukturen erlaubt, allerdings auf Kosten der Laufzeiteffizienz.

Der DEMETER-Ansatz setzt auch wieder auf der Ebene der Variantenbildung für Gesamtsysteme an. Durch die Verwendung eines Propagation Patterns kann sich die Semantik einer Operation leicht an Strukturänderungen des Systems anpassen lassen. Allerdings wird wie bei dem HYPER/J-Ansatz eine Variantenbildung für Objekte direkt nicht unterstützt.

7.9 Komponenten

Für die Frage, was eine Komponente eigentlich ist, existieren sehr viele Anworten (vgl. [SGM02, Kapitel 11], [Gri98, Abschnitt 2.2.2], [And03, Abschnitt 3.1]). Die folgende Liste präsentiert eine Auswahl dieser Antworten:

- BOOCH ([Boo87]):
  „A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.“

[12] Eine Propagation-Direktive kann auch mehrere Endknoten definieren.

- NIERSTRASZ und DAMI ([ND95]):
  „A software component is a static abstraction with plugs.“
- SAMETINGER ([Sam97]):
  „components = objects + something“
- HARRIS zitiert in [OHE97]:
  „A component is a piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability.“
- CZARNECKI und EISENECKER ([CE00]):
  „We define software components simply as building blocks from which different software systems can be composed. [...] We want them to be plug-compatible by design and to be combinable in as many ways as possible. We want to minimize code duplication and maximum reuse.“
- SZYPERSKI ([SGM02]):
  „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“
- UML VERSION 1.5 ([Obj03]):
  „A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component is typically specified by one or more classifiers (e.g., implementation classes) that reside on it, and may be implemented by one or more artefacts (e.g., binary, executable, or script files).“
- ANDRESEN ([And03]):
  „Eine Software-Komponente ist ein eigenständiges Artefakt eines Software-Systems, welche über spezifisches Wissen verfügt und gemäß ihrer Spezifikation über ein oder mehrere Schnittstellen mit anderen Software-Komponenten und -Systemen kommunizieren kann. Das Wissen einer Software-Komponente repräsentiert ein Konzept eines Geschäftsfeldes. Eine Komponente kann verpackt und unter Berücksichtigung eines Komponenten-Modells als autonome, wiederverwendbare Einheit verteilt werden.“

Als ein Beispiel für ein Komponentenmodell soll hier überblicksartig die J2EE-Architektur[13] beschrieben werden. Abb. 131 (S. 247) zeigt die Laufzeitumgebung einer EJB-Komponente[14]. Der EJB-Server stellt die Laufzeitumgebung für die EJB-Container dar und schirmt diese von den Besonderheiten des jeweiligen Betriebssystems ab. Er nimmt z.B. die Prozeß- und Thread-Verwaltung vor und ist für das *Pooling* gemeinsam genutzter Resourcen, wie Netzwerk- und Datenbankverbindungen, zuständig. Der EJB-Container stellt die Infrastruktur für die EJB-Komponenten zur Verfügung. Hierzu zählen Transaktionsverwaltung, Sicherheitsdienste und auch der Persistenzdienst. Ein EJB-Connector ist für die Anbindung externer Systeme, z.B. ERP-Systeme[15], verantwortlich. EJB-Komponenten sind *Business*-Komponenten. Ein Beispiel für eine EJB-Komponente ist ein Bankkonto oder ein Warenkorb. Für eine EJB-Komponente wird die nach außen sichtbare Schnittstelle über das Home- und das Remote-Interface spezifiziert. Nur über diese Schnittstellen ist der Zugriff auf die Komponente möglich. Die Konfiguration einer EJB-Komponente erfolgt durch den *Deployment*-Prozeß. Der zugehörige EJB

[13] J2EE: JAVA 2 Enterprise Edition
[14] EJB: Enterprise JavaBean
[15] ERP: Enterprise Resource Planning

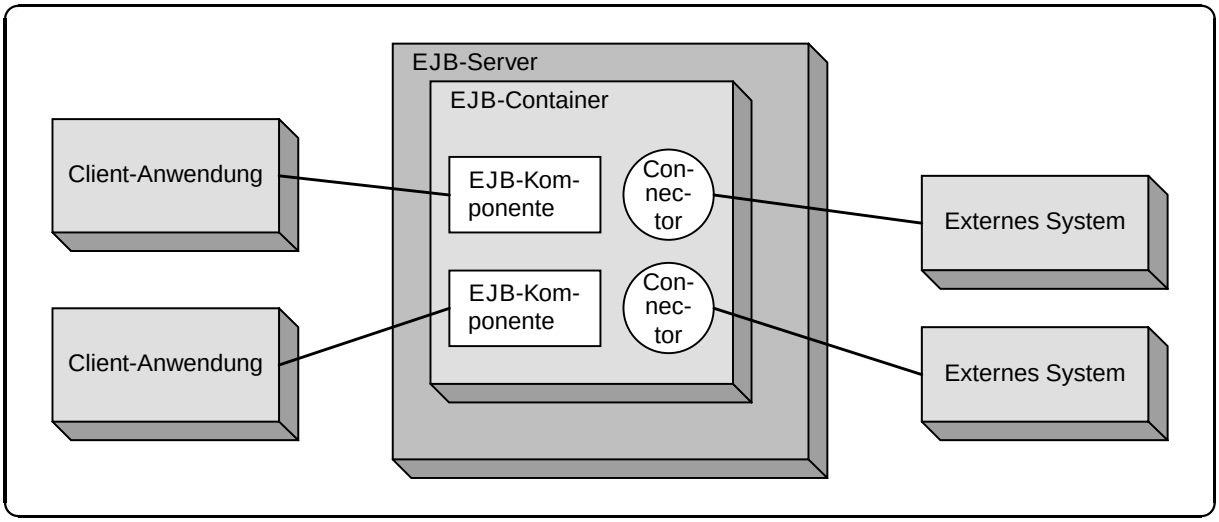


Abb. 131 Die Laufzeitumgebung einer EJB-Komponente (vgl. [And03, S. 263])

Deployment Descriptor ist eine XML-Datei, die beispielsweise Attribute zur Beschreibung des Transaktionsverhaltens oder Sicherheitsaspekte einer EJB-Komponente festlegt.

Die Zielsetzung des Komponentenkonzepts liegt in einer Bereitstellung wiederverwendbarer Bausteine. Diese können dann zur Konstruktion größerer Systeme eingesetzt werden, indem die ausgewählten Bauteile *zusammengesteckt* werden. Bei Komponentenmodellen steht die Umkonfiguration von Bauteilstrukturen zur Laufzeit nicht im Vordergrund. Daher werden die Anforderungen AV4 bis AV6 eines Variantenmodells (vgl. S. 222) nicht explizit unterstützt.

7.10 Kapitelzusammenfassung

In diesem Kapitel wurden alternative Ansätze zur Variantenbildung vorgestellt. Alle Ansätze können zwar prinzipiell benutzt werden, um die Anforderungen AV1 bis AV6 an ein Variantenmodell (vgl. S. 222) umzusetzen, allerdings ist dies nur mit erheblichen Aufwand möglich, da keiner der Ansätze für alle Anforderungen eine explizite Unterstützung anbietet. In Tab. 3 werden die besprochenen Ansätze gegenübergestellt. Ein Tabelleneintrag sagt aus, ob für die Erfüllung der entsprechenden Anforderung eine explizite Unterstützung in dem jeweiligen Ansatz vorhanden ist oder nicht. Bei der Bewertung von Hyperspaces und DEMETER ist zu beachten, daß bei diesen Ansätzen die Variantenbildung von Anwendungssystemen die Hauptzielrichtung darstellt. Die vorhandenen Konzepte zur Komposition von Kontrollflüssen decken jedoch die Anforderungen AV4 bis AV6 nicht explizit und vollständig ab. Daher sind die zugehörigen Tabelleneinträge im Sinne eines *eher nein* zu interpretieren.

Ansatz / Anforderung	AV1	AV2	AV3	AV4	AV5	AV6
Kompositionsbeziehung	ja	ja	ja	nein	nein	nein
Composite-Muster	ja	ja	ja	nein	nein	nein
Vererbungskonzept	nein	nein	ja	ja	ja	ja
MIXINS und TRAITS	nein	nein	ja	ja	ja	ja
RONDO	nein	nein	ja	ja	ja	ja
Hyperspaces	nein	nein	nein	nein	nein	nein
DEMETER	ja	ja	nein	nein	nein	nein
Komponenten	ja	ja	ja	nein	nein	nein

Tab. 3 Vergleich der Ansätze zur Variantenbildung

Für die Ansätze aus dem Vererbungsumfeld kommt die Einschränkung hinzu, daß sie die Variantenbildung einer Klasse über die *ist-ein*-Beziehung realisieren und so aus konzeptueller Sicht nicht zu dem Variantenmodell aus Abschnitt 6.2 (S. 221) passen, da dieses auf der *besteht-aus*-Beziehung aufbaut. Dafür besitzen diese Ansätze aber über die Methodenredefinition ein integriertes Konzept zur Anpassung von Kontrollflüssen in den abgeleiteten Klassen.

Im nächsten Kapitel wird ein neues Konzept zur Realisierung von Variantenmodellen vorgestellt, das die Anforderungen AV1 bis AV6 explizit unterstützt.

Kapitel 8

Ein neues Konzept zur Realisierung von Variantenmodellen

8.1 Einleitung

Für die Umsetzung der Anforderungen AV1 bis AV6 (vgl. S. 222) spielen die folgenden Aspekte eine zentrale Rolle:

- Die generelle Strukturbeschreibung eines Chassis-Objekts.
  Hier geht es um die Spezifikation, aus welchen Bauteilen (Chassis-Komponentenobjekte bzw. Komponentenobjekte) das Chassis-Objekt aufgebaut ist.
- Der globale Kontrollfluß von Operationen.
  Entsprechend der aktuellen Konfiguration sind von den einzelnen, extern aufrufbaren Operationen unterschiedliche Komponentenobjekte betroffen.
- Das Konfigurationswissen.
  Über das Konfigurationswissen wird festgelegt, welche Kontrollflüsse sich ändern, wenn Komponenten ein- oder ausgebaut werden.

Die ersten beiden Aspekte werden in der Systemanalysephase direkt erfaßt. Die Gesamtobjektstruktur wird über das statische Modell beschrieben, während die aus den Anwendungsfällen abgeleiteten Sequenzdiagramme die globalen Kontrollflüsse spezifizieren. Das Konfigurationswissen ist in der Menge der Sequenzdiagramme enthalten. Durch die Transformation des Analysemodells in das Entwurfsmodell und die Implementierung geht die explizite Darstellung dieser Aspekte jedoch verloren. Die Kompositionsbeziehungen, die im Klassendiagramm noch in einer globalen Sicht dargestellt sind, werden auf die einzelnen Klassen verteilt, genauso wie die globalen Kontrollflüsse aus den Sequenzdiagrammen auf die Operationen der Klassen abgebildet werden. Das Konfigurationswissen ist typischerweise durch `if`- oder `switch`-Anweisungen in die Operationen integriert.

Für die systematische Umsetzung der Anforderungen AV1 bis AV6 wird daher ein neues Realisierungskonzept entwickelt, das eine Trennung dieser Aspekte vorsieht:

- Die Funktionalität einer in der Analysephase ermittelten Klasse wird über ihre ***korrespondierende*** Klasse beschrieben. Diese Klasse besitzt weder Kompositionsbeziehungen zu

den anderen Klassen noch eine Beschreibung globaler Kontrollflüsse bzw. des Konfigurationswissens.

- Die Beziehungen zwischen den einzelnen Objekten und die globalen Kontrollflüsse werden über ***Script***-Objekte realisiert.
- Das Konfigurationswissen wird in einer eigenen ***Configuration***-Klasse beschrieben. Dort wird festgelegt, welche Kontrollflüsse beim Ein- oder Ausbau von Komponenten auszutauschen sind.

Um die Systementwicklungsphasen zu integrieren, wird ein Generierungsprozeß verwendet. Ausgehend von einer Strukturbeschreibung des Variantenmodells und den korrespondierenden Klassen werden alle benötigten Klassen und Interfaces erzeugt, die für eine lauffähige Version notwendig sind. Diese stellt den Ausgangspunkt für die manuelle Implementierung abgeleiteter Konfigurations- und Script-Klassen dar, die dann das modellspezifische Konfigurationswissen und die modellspezifischen globalen Kontrollflüsse realisieren. Dieser manuelle Schritt ist notwendig, da die Strukturbeschreibung keine Spezifikation der globalen Kontrollflüsse enthält.

Nach einer Beschreibung des Analyseprozesses für Variantenmodelle (Abschnitt 8.2 (S. 251)) wird daraus ein Entwurfsmodell abgeleitet (Abschnitt 8.3 (S. 266)), das die Grundlage für den späteren Generierungsprozeß bildet. Es folgt die Beschreibung der Software-Architektur für die Generierung der Variantenmodelle (Abschnitt 8.4 (S. 295)). Den inhaltlichen Abschluß des Kapitels bildet die Darstellung der Integration des Variantenmodells in den Software-Entwicklungsprozeß aus der Sicht des Anwendungsentwicklers (Abschnitt 8.5 (S. 311)).

8.2 Analysemodell VMA

8.2.1 Das statische Modell

8.2.1.1 Chassis- und Komponentenklassen

In einem ersten Schritt sind für die verschiedenen Chassis- und Komponentenklassen die nach außen sichtbaren Operationen zu identifizieren und modellieren. Bei den Komponentenklassen umfaßt dies alle Operationen, die prinzipiell verfügbar sind, unabhängig davon, ob sie später bei der Integration eines Komponentenobjekts in ein konkretes Chassis-Objekt auch nutzbar sind. Für eine Chassis-Klasse werden diejenigen Grundoperationen modelliert, die ein Chassis-Objekt besitzt, wenn keines der Komponentenobjekte eingebaut ist.

In Abb. 132 sind die Chassis-Klassen **Auto** und **RadioTester** sowie die Komponentenklassen **Radio**, **Tuer** und **Schiebedach** dargestellt.[1]

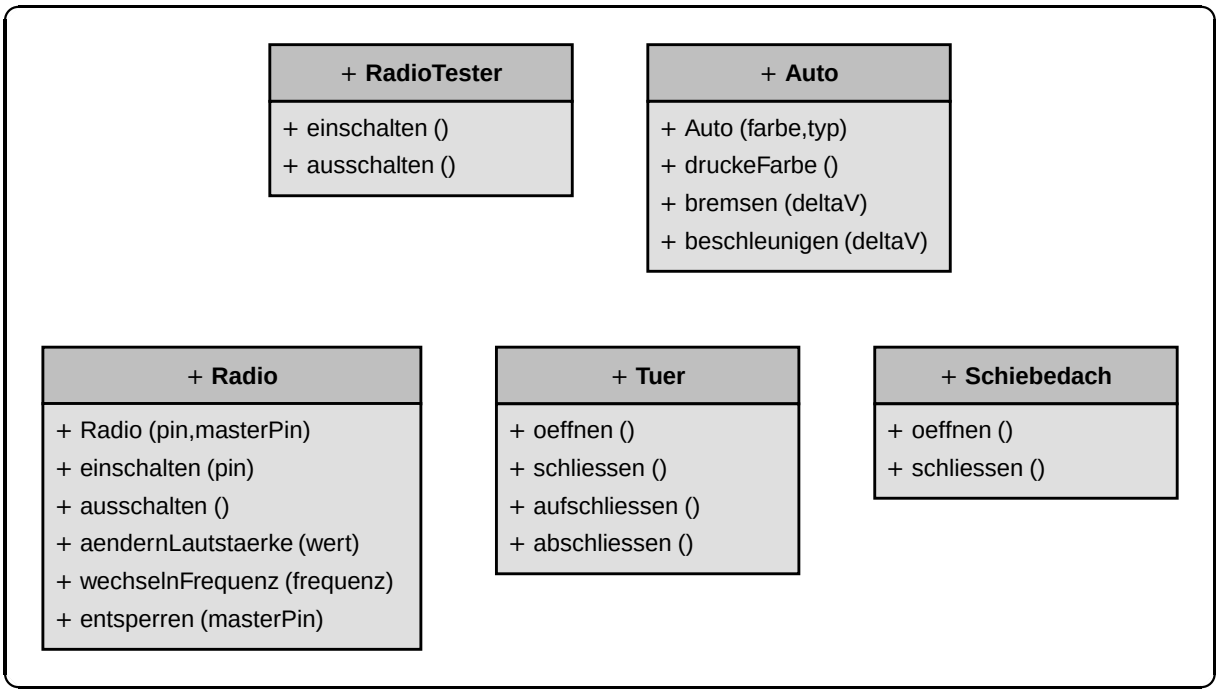


Abb. 132 Modellierung des Autobeispiels – Ergebnis des ersten Schritts

Innerhalb des zweiten Schritts wird festgelegt, welche Komponenten in welches Chassis eingebaut werden können. Ein Radio kann in ein Auto und einen Radiotester integriert werden, während die Komponenten Schiebedach und Tür nur in das Auto direkt einbaubar sein sollen.

Im nächsten Schritt sind die Operationen hinzuzufügen, die für die Verwaltung eines Variantenmodells sinnvoll sind:

- Das Ein- und Ausbauen einer Komponente,
- die Abfrage, ob in einem Chassis-Objekt eine bestimmte Komponente bereits eingebaut ist,

[1] Es wurden nur so viele Operationen modelliert, daß sich später anhand dieses Beispiels die entwickelten Konzepte demonstrieren lassen.

- die Abfrage, ob ein Komponentenobjekt in (irgendeinem) Chassis-Objekt eingebaut ist, sowie
- das Löschen von Chassis- und Komponentenobjekten.

Das Ergebnis der Schritte 2 und 3 ist in Abb. 133 (S. 253) zu sehen. Die Zuordnung der Komponentenobjekte zu den Chassis-Objekten führt zu den Assoziationen mit den entsprechenden Kardinalitäten.

Der letzte Schritt betrifft die Anforderung, daß die nutzbare Funktionalität eines Komponentenobjekts von dem Chassis-Objekt abhängen kann, in das es eingebaut ist. Dies läßt sich modellieren, indem die jeweils nutzbaren Operationen an der Schnittstelle der Chassis-Objekte angeboten werden und zusätzlich die Restriktion gelten soll, daß der Zugriff auf die Operationen der Komponentenobjekte nur über die Chassis-Objekte erfolgt. Das Chassis-Objekt hat dann die Aufgabe, eine entsprechende Fehlermeldung zu erzeugen, falls versucht wird, auf eine Komponente zuzugreifen, die zur Zeit nicht im Chassis-Objekt installiert ist. Die entstehende Struktur der Chassis-Klassen **Auto** und **RadioTester** ist in Abb. 134 (S. 254) dargestellt.

Für das Autobeispiel liegt somit ein Variantenmodell vor, welches zwei Ausgangsklassen besitzt, **Auto** und **RadioTester**.

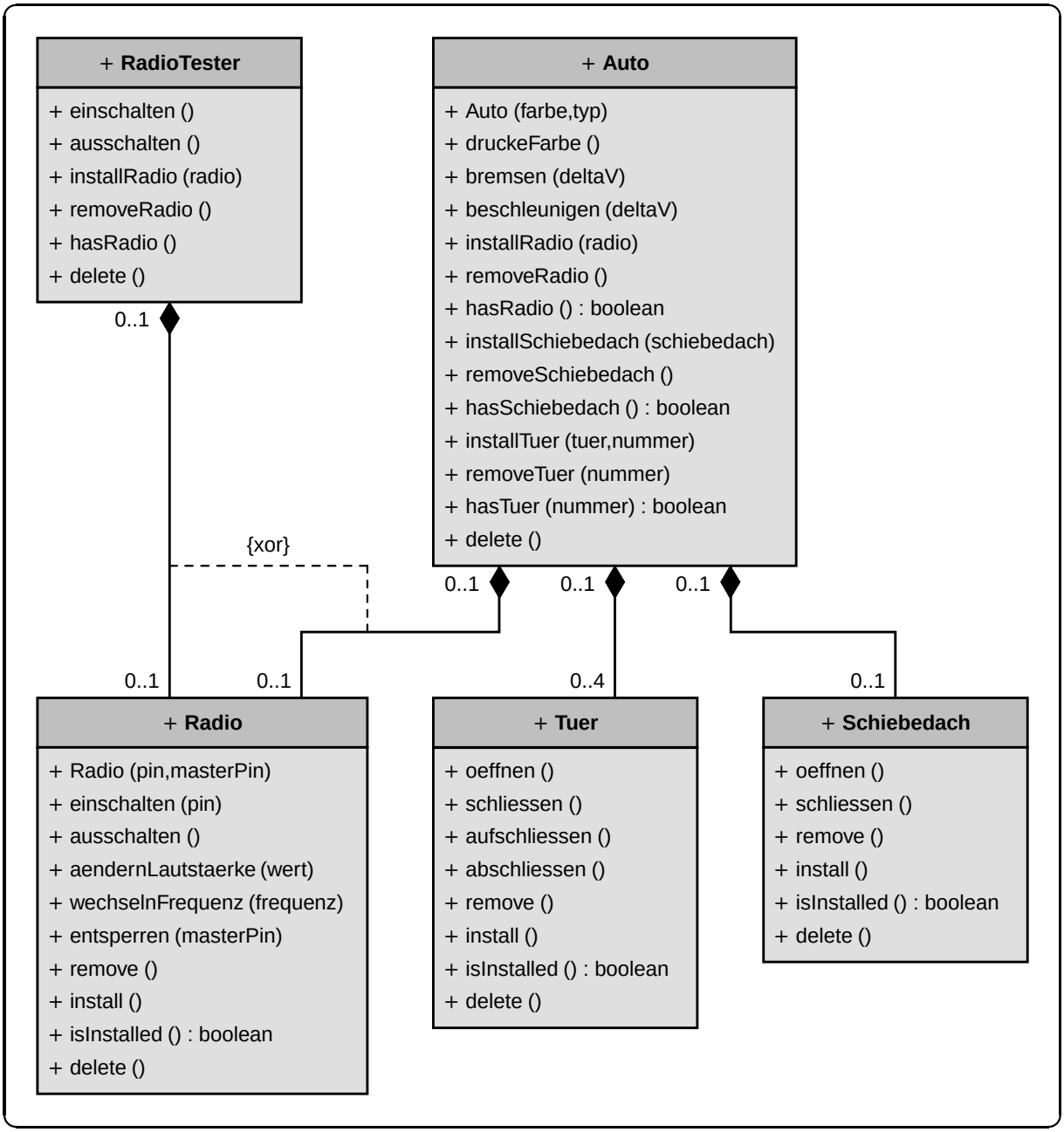


Abb. 133 Modellierung des Autobeispiels – Ergebnis der Schritte 2 und 3

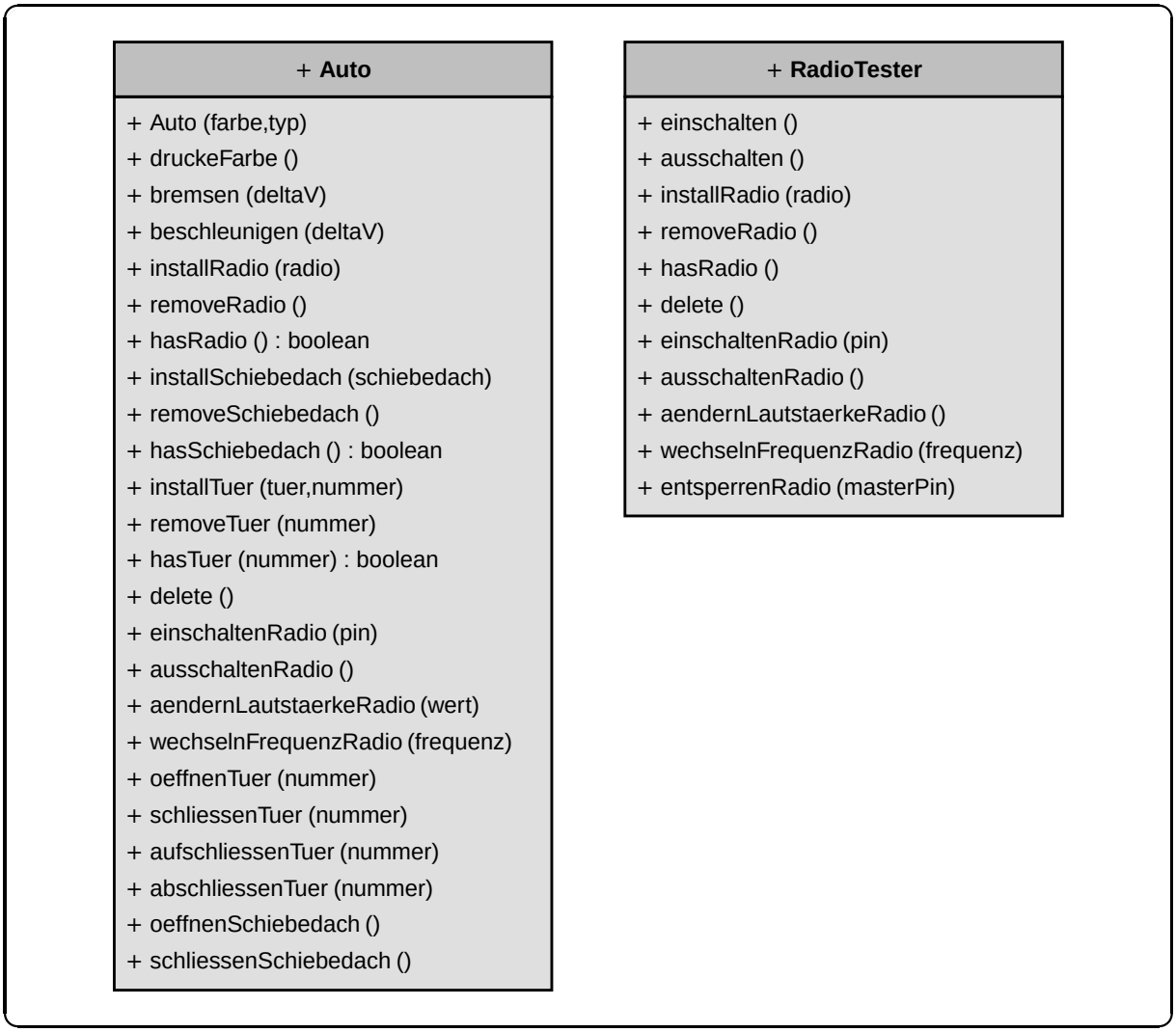


Abb. 134 Modellierung des Autobeispiels – Ergebnis von Schritt 4

8.2.1.2 Die Verwendung von mehr als zwei Hierarchieebenen

Die im vorherigen Abschnitt beschriebenen Schritte zur Definition eines Variantenmodells können rekursiv angewendet werden, um ein Modell mit mehreren Hierarchiestufen zu definieren. Da ein Variantenmodell mehrere Chassis-Klassen besitzen kann, entsteht insgesamt ein ***azyklischer Graph*** mit den folgenden Eigenschaften:

- Die Knoten des Graphen sind die Bauteilklassen.
- Eine Kante von **A** nach **B** bedeutet, daß das Bauteil **B** in **A** eingebaut werden kann.
- Die Knoten des Graphen, die keine Vorgänger haben (Ausgangsknoten), definieren Bauteilklassen, die nur die Chassis-Funktionalität besitzen.
- Die inneren Knoten stellen Bauteilklassen dar, die sowohl über die Chassis- als auch die Komponentenfunktionalität verfügen.
- Bauteile ohne Nachfolger im Graphen (Blattknoten) sind reine Komponentenklassen.

Die Ausgangsknoten des Variantenmodells sind die ***Chassis-Klassen***. Die inneren Klassen eines Variantenmodells werden in der Folge als ***Chassis-Komponentenklassen*** bezeichnet. Der Begriff ***Komponentenklasse*** wird ausschließlich für die Blattknotenklassen des Variantenmodells verwendet. Die direkten Nachfolger eines Knotens des Graphen werden als ***Unterkomponenten*** bezeichnet. Dabei kann es sich um Chassis-Komponenten- oder Komponentenklassen handeln.

In Abb. 135 ist ein Variantenmodell mit zwei Chassis-Klassen dargestellt.

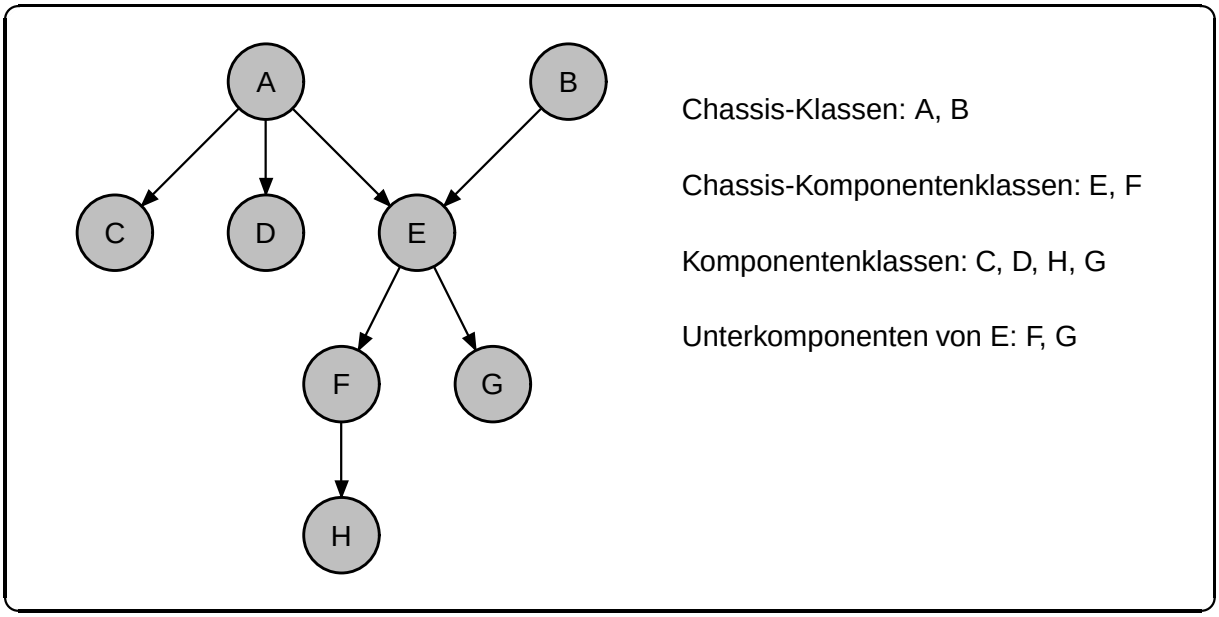


Abb. 135 Graphdarstellung eines Variantenmodells

Für das Autobeispiel soll ein **Radio**-Objekt jetzt die Eigenschaft erhalten, daß ein optionaler CD-Spieler eingebaut werden kann. Die Funktionalität der Klasse **CdSpieler** ist in Abb. 136 (S. 256) beschrieben, die Graphdarstellung des resultierenden Variantenmodells in Abb. 137 (S. 256).

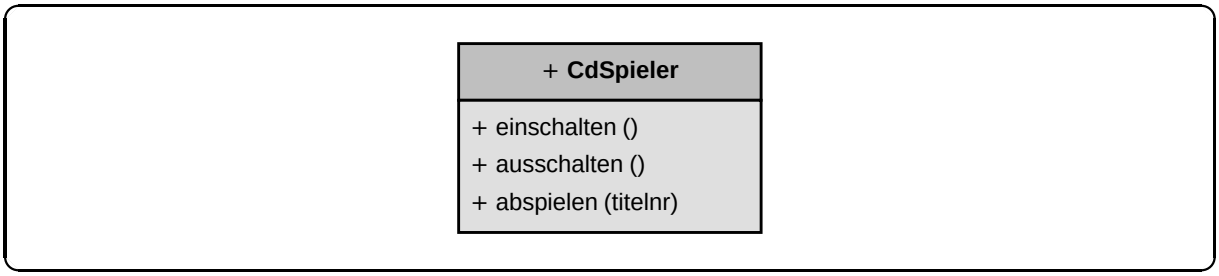


Abb. 136 Die Funktionalität der Klasse **CdSpieler**

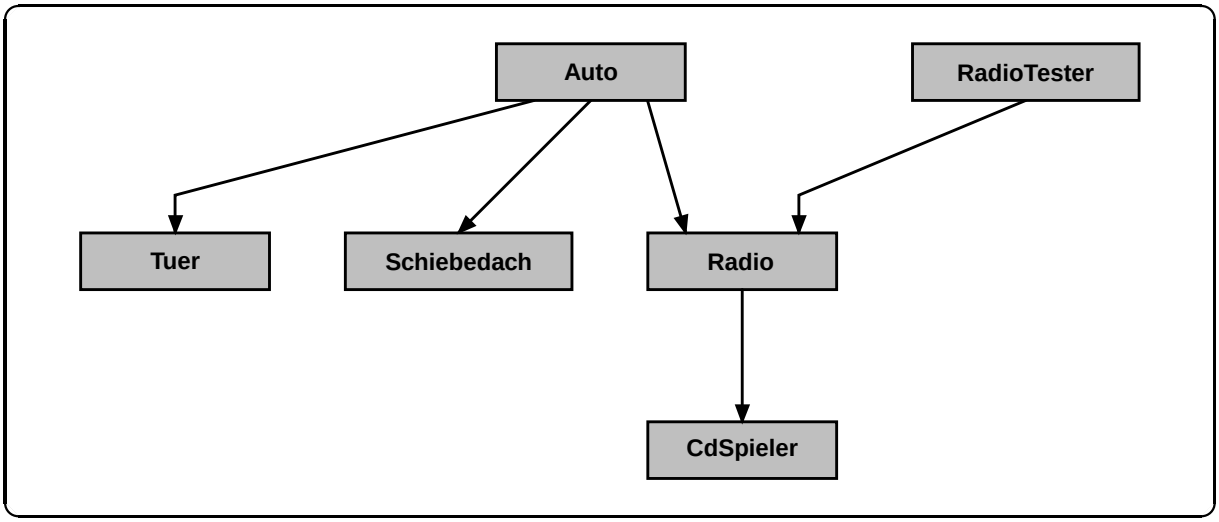


Abb. 137 Graphdarstellung des Variantenmodells eines Autos

Die Operationen `einschalten` und `ausschalten` des CD-Spielers sollen an der Schnittstelle des Radios nicht ansprechbar sein. Stattdessen wird der CD-Spieler mit dem Ein- und Auschalten des Radios seinerseits ein- und ausgeschaltet. Die Operation `abspielen` ist dagegen an der Schnittstelle des Radios aufrufbar, das aus der Sicht des CD-Spielers die Rolle des Chassis-Objekts übernimmt. Da das Radio selbst wieder in ein Auto eingebaut werden kann, wird auch an der Autoschnittstelle die Abspieloperation sichtbar. Als Operationsname wurde hier `abspielenCdSpieler` gewählt. Das Radio erhält in seiner Funktion als Chassis-Objekt für den CD-Spieler die zusätzlichen Operationen `installCdSpieler`, `removeCdSpieler` und `hasCdSpieler`. Insgesamt ergibt sich damit in Schritt 4 der Modellierung des Autobeispiels die in Abb. 138 (S. 257) gezeigte Struktur, wobei auf eine Darstellung der Klasse **RadioTester** verzichtet wurde, da sie sich bezüglich ihres prinzipiellen Aufbaus nicht von der Radioklasse unterscheidet.

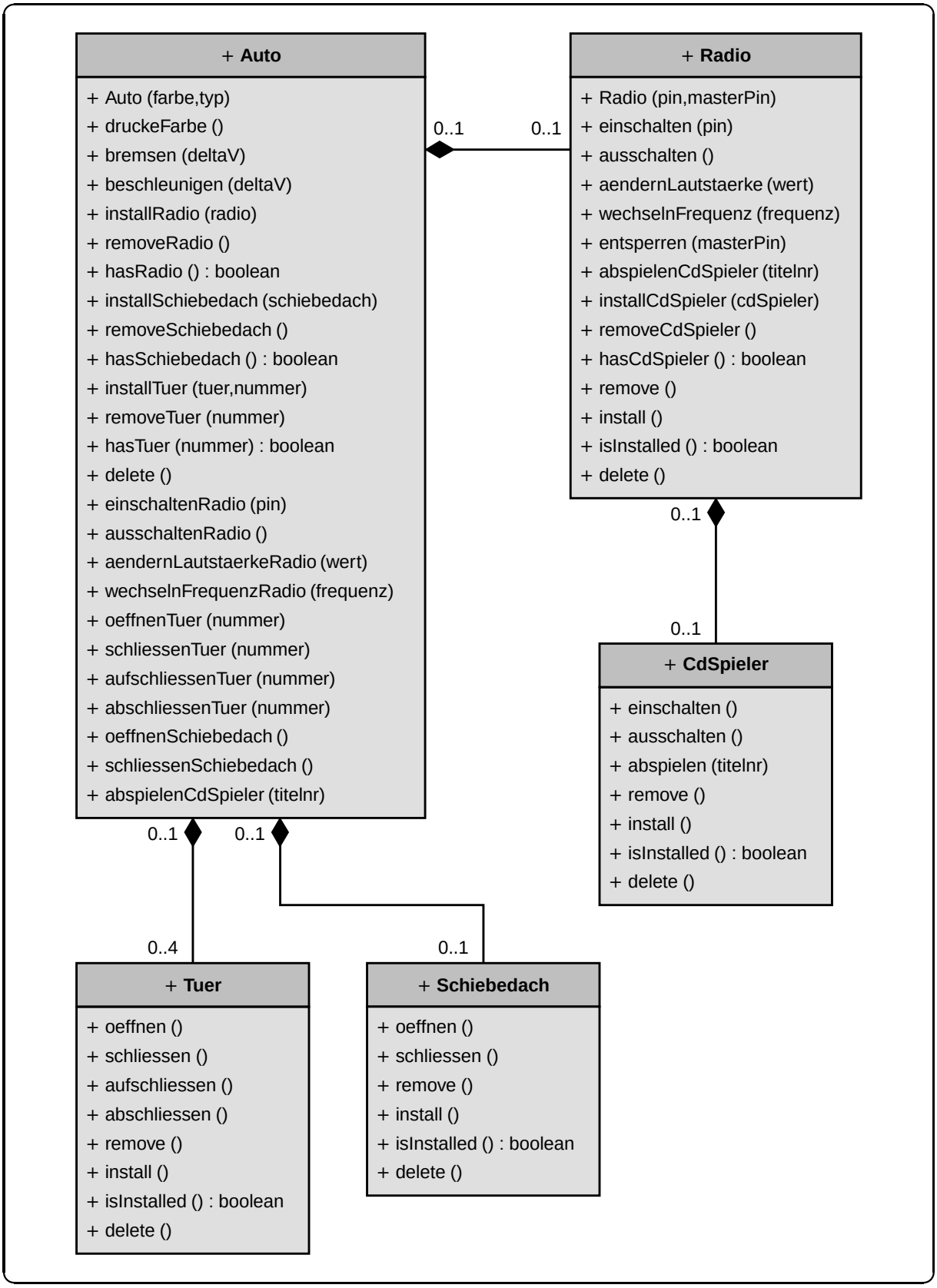


Abb. 138 Modellierung des Autobeispiels mit optionalem CD-Spieler – Ergebnis von Schritt 4

8.2.2 Das dynamische Modell für zwei Hierarchieebenen

8.2.2.1 Einteilung der Operationen in Kategorien

Die Anforderungen AV5 und AV6 (vgl. S. 222) betreffen das dynamische Modell. Auf der Analyseebene muß spezifiziert werden, wie sich der Kontrollfluß der Operationen einer Chassis-Klasse durch den Ein- und Ausbau von Komponentenobjekten verändert.

Zunächst soll der einfachste Fall eines Variantenmodells mit zwei Hierarchieebenen betrachtet werden. Die Operationen der Komponentenklassen und der Chassis-Klasse lassen sich in verschiedene Kategorien einordnen:

- Komponentenklasse:
  - **Primäroperationen**
    Dieses sind die Operationen einer Komponentenklasse, die die semantische Funktionalität eines Komponentenobjekts beschreiben. Die zugehörigen Operationen der Komponentenklassen **Radio**, **Tuer** und **Schiebedach** sind in Abb. 132 (S. 251) beschrieben.
  - **Konfigurationsoperationen**
    Diese Menge besteht aus den Operationen `install` und `remove`. Sie werden aufgerufen, wenn ein Komponentenobjekt in ein Chassis-Objekt ein- bzw. ausgebaut wird.
  - (Sonstige)[2] **Verwaltungsoperationen**
    Hierunter fallen die Operationen `isInstalled` und `delete`.
- Chassis-Klasse:
  - **Primäroperationen**
    Dieses sind die Operationen einer Chassis-Klasse, die unabhängig vom vorliegenden Variantenmodell existieren. Für die Chassis-Klasse **Auto** sind es die in Abb. 132 (S. 251) dargestellten Operationen.
  - **Sekundäroperationen**
    Wenn das Variantenmodell die Komponentenklassen **CompClass**$_1$ bis **CompClass**$_n$ besitzt, dann besteht die Menge der Sekundäroperationen der Chassis-Klasse aus allen Primäroperationen von **CompClass**$_1$ bis **CompClass**$_n$, die gemäß der Anforderung AV4 (vgl. S. 222) an der Schnittstelle zur Chassis-Klasse sichtbar sein sollen.
  - **Konfigurationsoperationen**
    Für jede Komponentenklasse **CompClass**$_i$ ($i \in \{1,\ldots,n\}$) existieren die Konfigurationsoperationen `installCompClass`$_i$ und `removeCompClass`$_i$.
  - (Sonstige) **Verwaltungsoperationen**
    Neben der `delete`-Operation sind dies für ($i \in \{1,\ldots,n\}$) die `hasCompClass`$_i$-Operationen.

8.2.2.2 Zustandsautomaten

Alle Komponentenobjekte besitzen einen globalen Zustandsautomaten, der in Abb. 139 (S. 259) dargestellt ist.

[2] Die Konfigurationsoperationen sind natürlich auch Verwaltungsoperationen. Es ist hier aber sinnvoll, sie von den restlichen Verwaltungsoperationen abzugrenzen.

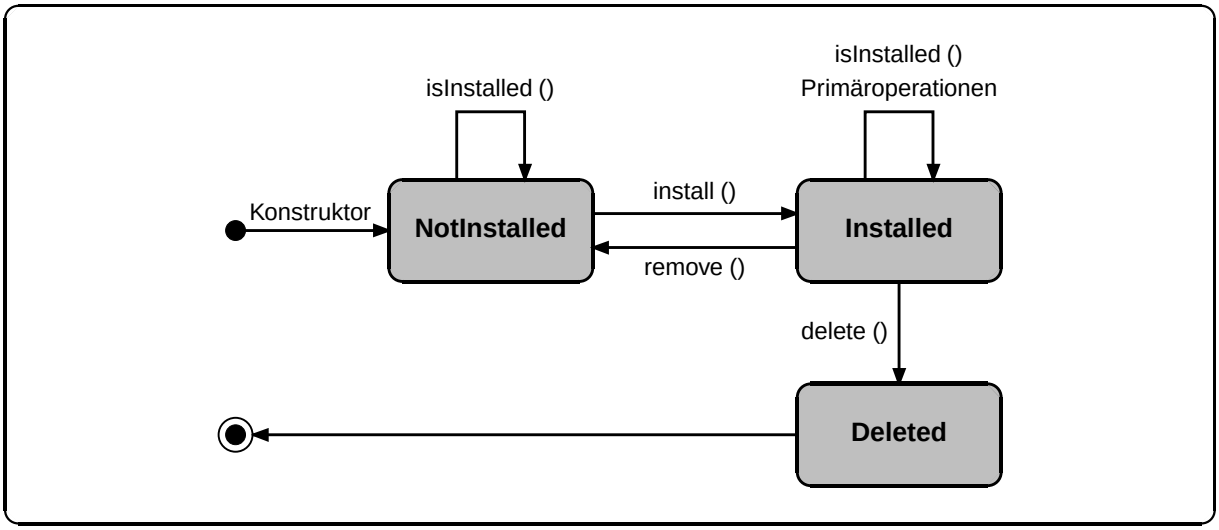


Abb. 139 Der globale Zustandsautomat eines Komponentenobjekts

Die Primäroperationen sind nur ausführbar, wenn das Komponentenobjekt in ein Chassis-Objekt eingebaut ist. Im *Installed*-Zustand kann dann für eine Komponentenklasse ein weiterer lokaler Zustandsautomat existieren, was bei der Modellierung technischer Objekte sehr häufig vorkommt, da bestimmte Operationen nur in bestimmten Zuständen erlaubt sind. Bei einem Radioobjekt wird man beispielsweise erwarten, daß mit Ausnahme der Operation `einschalten` alle anderen Operationen nur im Zustand *Eingeschaltet* erlaubt sind. Da alle Komponentenklassen den globalen Zustandsautomaten mit derselben Semantik besitzen, bietet es sich an, die Operationen `remove`, `install`, `isInstalled` und `delete` in eine abstrakte Oberklasse **BaseComponent** zu übernehmen. Für die Komponentenklassen aus Abb. 133 (S. 253) ergibt sich damit die in Abb. 140 dargestellte Vererbungsstruktur.

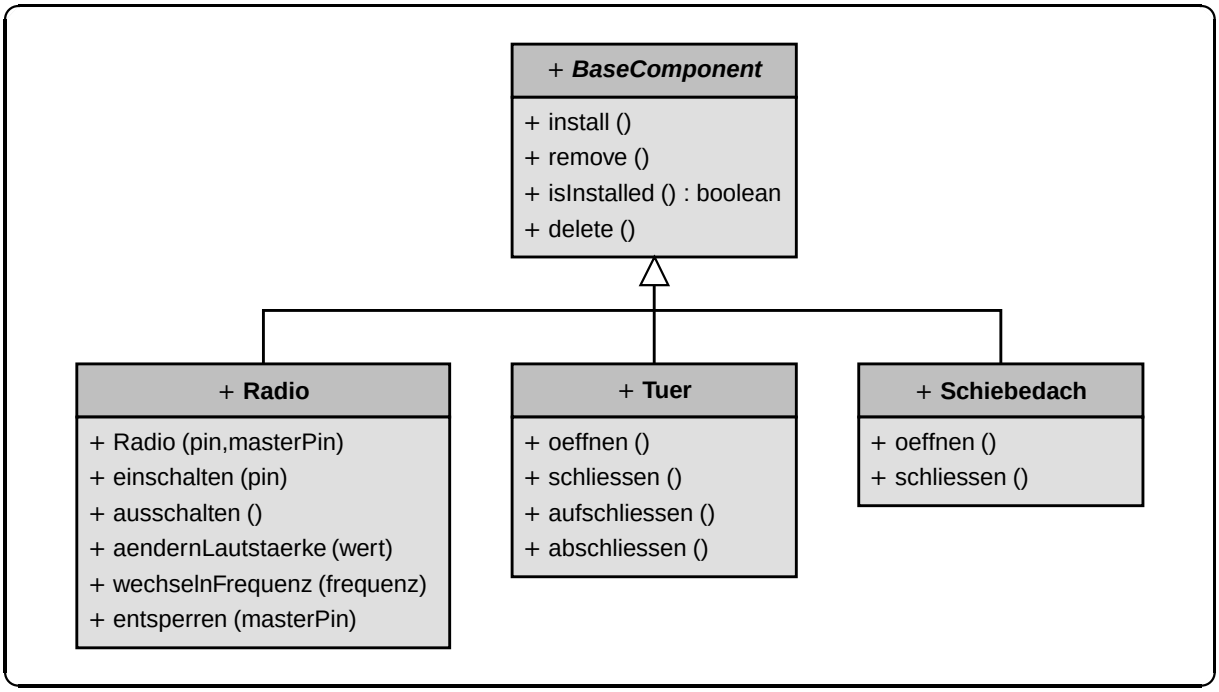


Abb. 140 Verwendung der abstrakten Klasse **BaseComponent**

8.2.2.3 Kontrollfluß zwischen den Objekten

Die Primär- und Sekundäroperationen stellen die Anwendungsfälle eines Chassis-Objekts dar. Für jeden Anwendungsfall ist jetzt zu analysieren, welcher Kontrollfluß zwischen dem Chassis- und seinen Komponentenobjekten in Abhängigkeit der vorliegenden Konfiguration entsteht. Abb. 141 zeigt das UML-Sequenzdiagramm für die Operation `abschliessenTuer` unter der Voraussetzung, daß Objekte der Komponentenklassen **Tuer**, **Radio** und **Schiebedach** im Auto installiert sind.

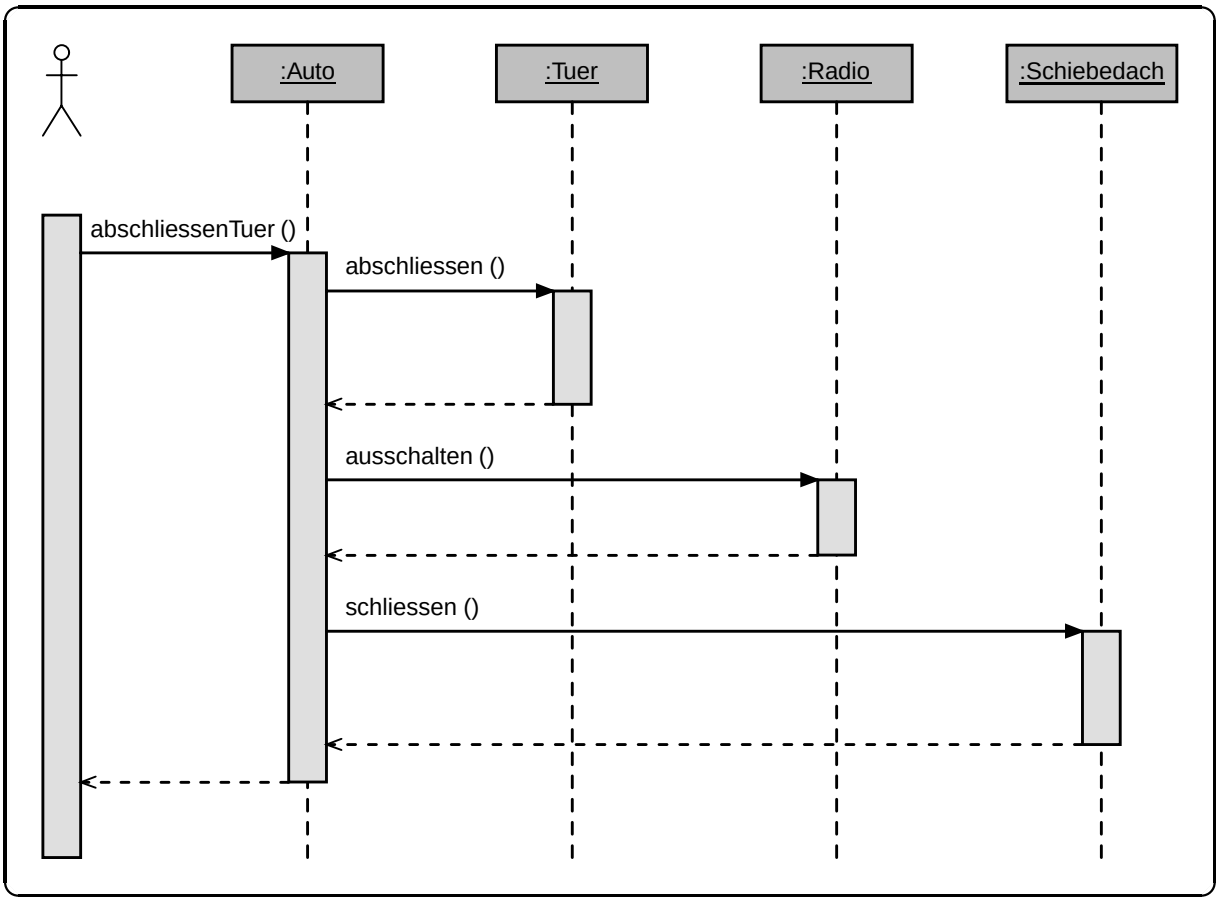


Abb. 141 Das UML-Sequenzdiagramm der Operation `abschliessenTuer`

Besitzt das Auto dagegen kein Schiebedach, dann fällt entsprechend beim Abschließen der Tür der Operationsaufruf `schliessen` weg.

Liegen für eine Chassis-Klasse n Komponentenklassen vor, so gibt es 2^n Konfigurationsvarianten, sofern alle Komponentenkombinationen erlaubt sind. Damit könnte es prinzipiell pro Primär- und Sekundäroperation 2^n verschiedene Kontrollflüsse geben. In der Regel werden aber nicht alle Operationen von der Konfiguration abhängen. Für das Autobeispiel ist es nur die Operation `abschliessen` der **Tuer**-Komponente, wie man der ***Kontrollflußmatrix*** aus Abb. 142 (S. 261) entnehmen kann. Ein Kreuz in einer Zeile bedeutet, daß bei der Ausführung einer Operation aus der zweiten Spalte von rechts die durch das Kreuz markierte Operation aufzurufen ist, sofern das zugehörige Komponentenobjekt installiert ist.

Um die Unabhängigkeit der Komponentenobjekte untereinander zu gewährleisten, dürfen zwi-

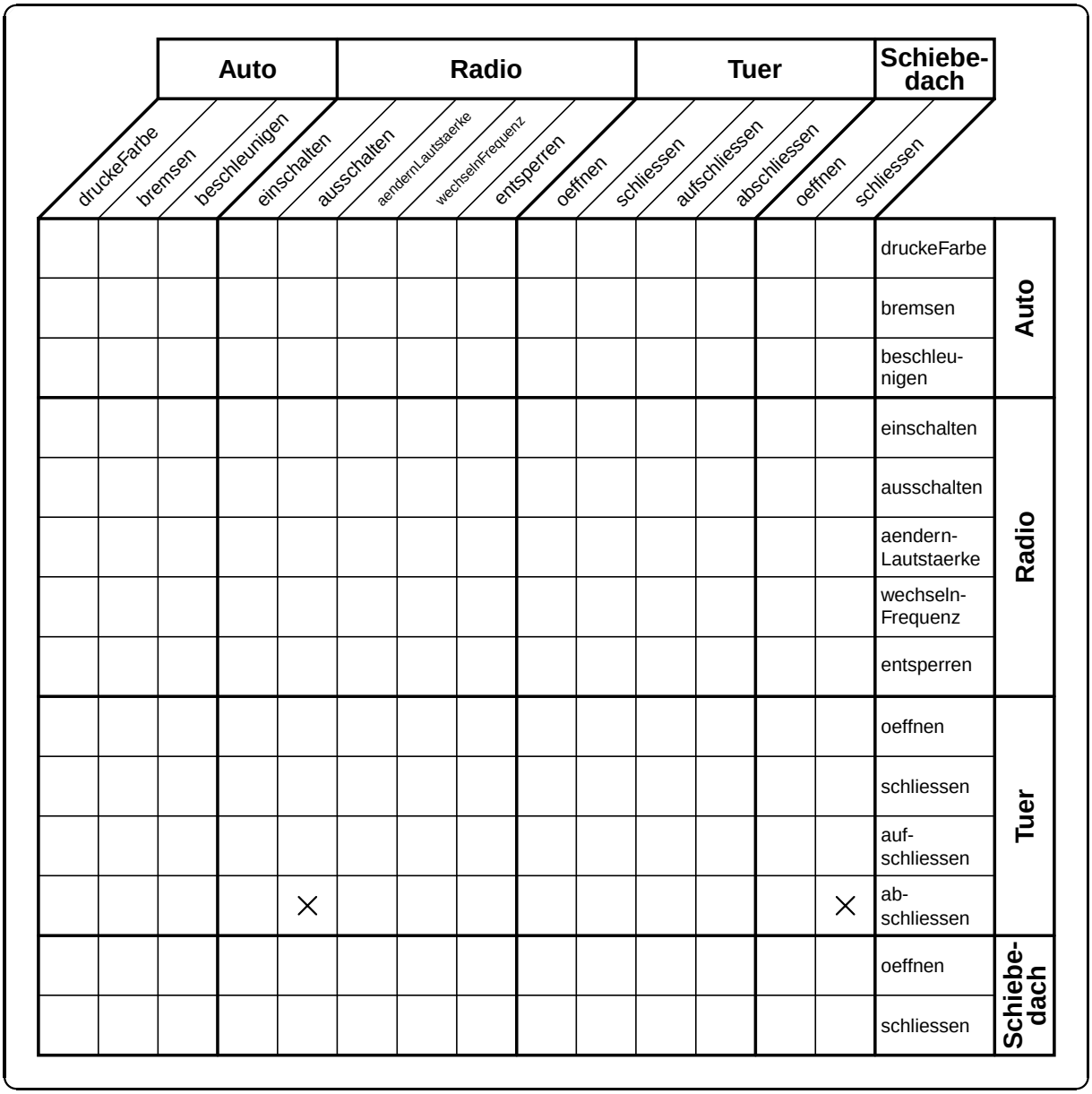


Abb. 142 Kontrollflußmatrix für das Autobeispiel

schen den Primäroperationen unterschiedlicher Komponentenobjekte keine direkten Aufrufbeziehungen bestehen. Die Steuerung des Kontrollflusses erfolgt vollständig durch die von außen aufgerufene Primär- oder Sekundäroperation des Chassis-Objekts.

8.2.3 Das dynamische Modell für mehr als zwei Hierarchieebenen

8.2.3.1 Einteilung der Operationen in Kategorien

Neben den Chassis- und Komponentenklassen existieren jetzt ***Chassis-Komponentenklassen***, die bezüglich ihrer Operationskategorien eine Kombination aus den Chassis- und Komponentenklassen darstellen (vgl. S. 258). Für eine Chassis-Komponentenklasse **CC** liegen folgende Operationskategorien vor:

- **Primäroperationen**
  Diese Operationen beschreiben die nach außen sichtbare semantische Funktionalität. Die zugehörigen Operationen für die Klasse **Radio** sind in Abb. 132 (S. 251) beschrieben.
- **Sekundäroperationen**
  In ein Chassis-Komponentenobjekt lassen sich Komponenten- und Chassis-Komponentenobjekte einbauen. Die Menge der entsprechenden Klassen sei **CompClass**$_i$ ($i \in \{1,...,n\}$) und **ChassisCompClass**$_j$ ($j \in \{1,...,k\}$). Damit besteht die Menge der Sekundäroperationen von **CC** aus allen Primäroperationen von **CompClass**$_i$ und **ChassisCompClass**$_j$, die gemäß Anforderung AV4 (vgl. S. 222) in **CC** sichtbar sein sollen. Für die Klasse **Radio** ist die einzige Sekundäroperation `abspielenCdSpieler`.
- **Konfigurationsoperationen**
  Für die Klassen **CompClass**$_i$ und **ChassisCompClass**$_j$ ($i \in \{1,...,n\}$, $j \in \{1,...,k\}$) existieren die Konfigurationsoperationen `installCompClass`$_i$ und `installChassisCompClass`$_j$. Dazu kommen die Operationen `install` und `remove`. Da die **Radio**-Klasse nur eine Komponentenklasse besitzt, besteht die Menge der Konfigurationsoperationen aus `installRadio`, `removeRadio`, `install` und `remove`.
- (Sonstige) **Verwaltungsoperationen**
  Neben den `isInstalled`- und `delete`-Operationen sind dies die `hasCompClass`$_i$- und `hasChassisCompClass`$_j$-Operationen ($i \in \{1,...,n\}$, $j \in \{1,...,k\}$). Für die **Radio**-Klasse liegen als Verwaltungsoperationen `isInstalled`, `delete` und `hasCdSpieler` vor.

Die Kategorisierung der Chassis- und Komponentenklassenoperationen ändert sich nicht.

8.2.3.2 Zustandsautomaten

Der globale Zustandsautomat eines Chassis-Komponentenobjekts besitzt gegenüber dem globalen Zustandsautomaten eines Komponentenobjekts (vgl. Abb. 139 (S. 259)) mehrere Änderungen: im *Installed*-Zustand sind neben den Primäroperationen jetzt auch die Sekundäroperationen erlaubt; außerdem kommen die Konfigurations- und Verwaltungsoperationen hinzu. Abb. 143 (S. 263) zeigt den globalen Zustandsautomaten eines Chassis-Komponentenobjekts. Ein lokaler Zustandsautomat für die Chassis-Komponentenklasse kann dann wieder durch eine Verfeinerung des *Installed*-Zustands modelliert werden.

Der globale Zustandsautomat der Komponentenobjekte ist durch die Verwendung von mehr als zwei Hierarchieebenen nicht betroffen.

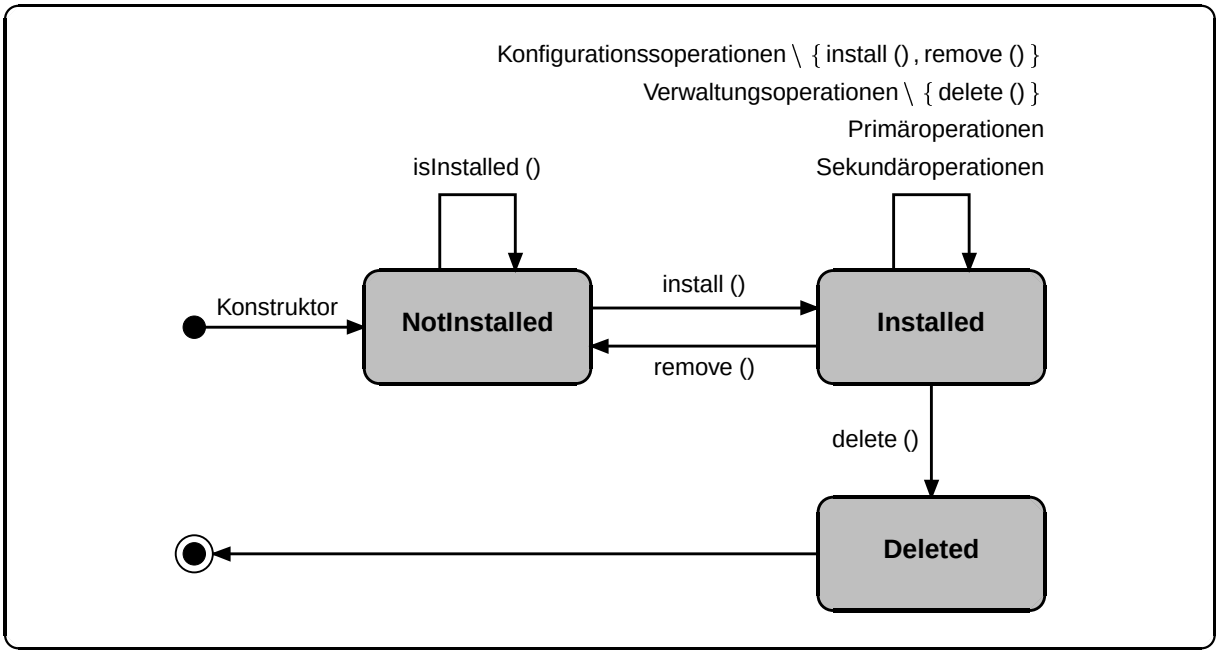


Abb. 143 Der globale Zustandsautomat eines Chassis-Komponentenobjekts

8.2.3.3 Kontrollfluß zwischen den Objekten

Auch bei mehr als zwei Hierarchieebenen werden die Anwendungsfälle über die Primär- und Sekundäroperationen des Chassis-Objekts definiert. Der Kontrollfluß einer derartigen Operation kann sich jetzt über alle Hierarchieebenen erstrecken. Ein Beispiel hierfür ist in dem Sequenzdiagramm aus Abb. 144 (S. 264) dargestellt. Das Abschließen der Tür führt zum Schließen des Schiebedachs und Ausschalten des Radios. Ist im Radio ein CD-Spieler installiert, so wird dieser ebenfalls ausgeschaltet.

Um ein Chassis-Objekt unabhängig von der aktuellen Konfiguration eines Chassis-Komponentenobjekts zu machen, dürfen zwischen dem Chassis-Objekt und den Unterkomponenten des Chassis-Komponentenobjekts keine direkten Aufrufbeziehungen bestehen. Im Beispiel aus Abb. 144 (S. 264) ruft daher das **Auto**-Objekt die `ausschalten`-Operation auf dem **Radio**-Objekt auf, welches dann seinerseits die `ausschalten`-Operation auf dem **CdSpieler**-Objekt aktiviert. Hält man die obige Regel für die Struktur der Aufrufbeziehungen ein, läßt sich die Kontrollflußmatrix für ein Chassis-Komponentenobjekt unabhängig von dessen Chassis-Objekt beschreiben. Die entsprechende Matrix für die Klasse **Radio** ist in Abb. 145 (S. 265) enthalten.

Die Primär- und Sekundäroperationen eines Chassis-Komponentenobjekts können damit wieder als eigenständige Anwendungsfälle aufgefaßt werden. Für jeden Anwendungsfall lassen sich in Abhängigkeit der aktuellen Konfiguration des Chassis-Komponentenobjekts ein oder mehrere Sequenzdiagramme spezifizieren.

Als Ergebnis dieser Kontrollflußstrukturierung läßt sich das Sequenzdiagramm aus Abb. 144 (S. 264) in zwei unabhängige Sequenzdiagramme zerlegen. Das erste Sequenzdiagramm beschreibt die Ausführung der Operation `abschliessenTuer` unabhängig davon, ob im Radio ein CD-Spieler installiert ist, und entspricht somit dem Sequenzdiagramm aus Abb. 141 (S. 260). Das zweite Sequenzdiagramm wird in Abhängigkeit der aktuellen Konfiguration der **Radio**-Klasse ausgewählt und definiert den lokalen Kontrollfluß der Operation `ausschalten`,

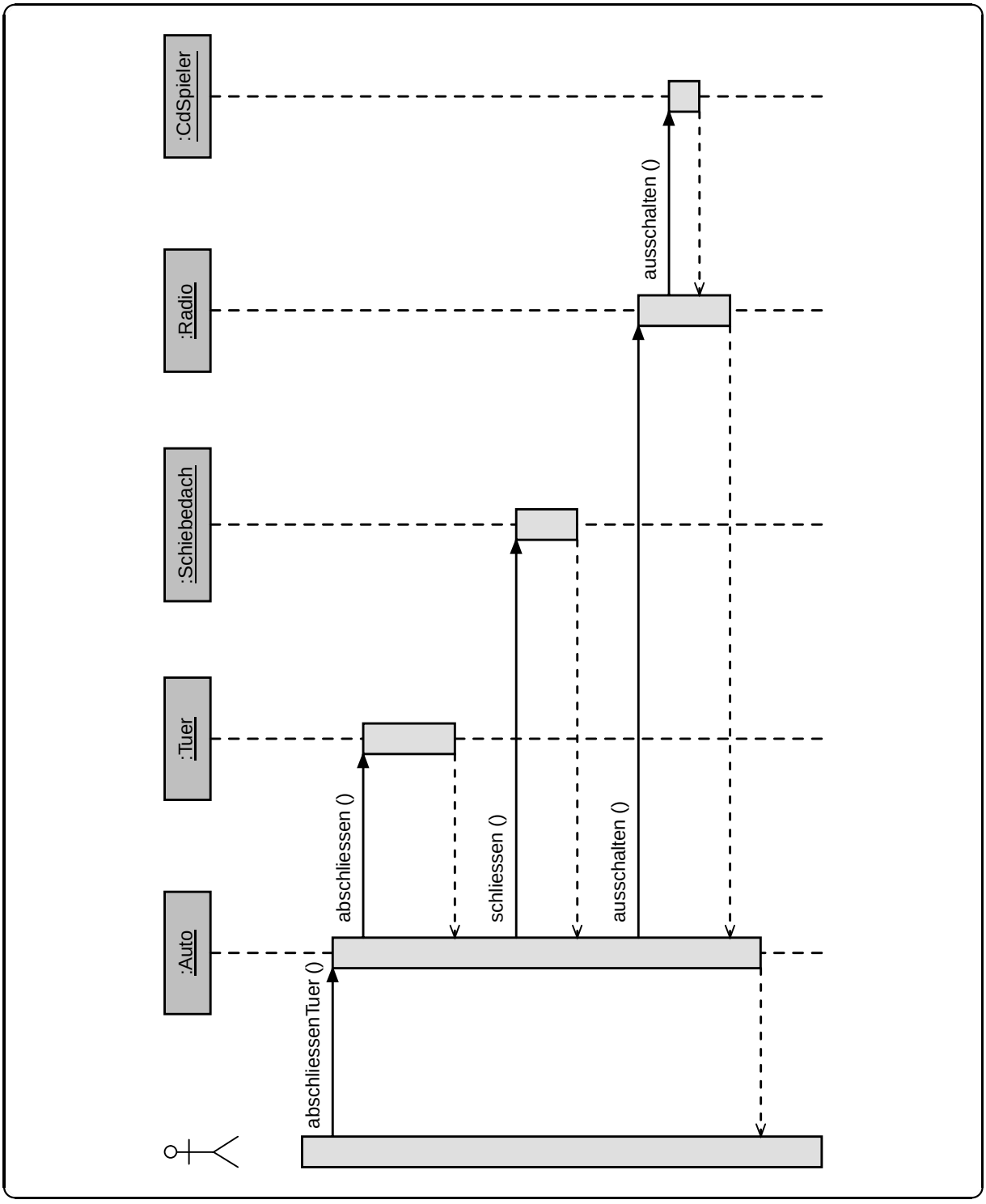


Abb. 144 Das Sequenzdiagramm der Operation `abschliessenTuer` bei einem installierten CD-Spieler

die von dem ersten Sequenzdiagramm während des Schliessens der Tür aktiviert wird.

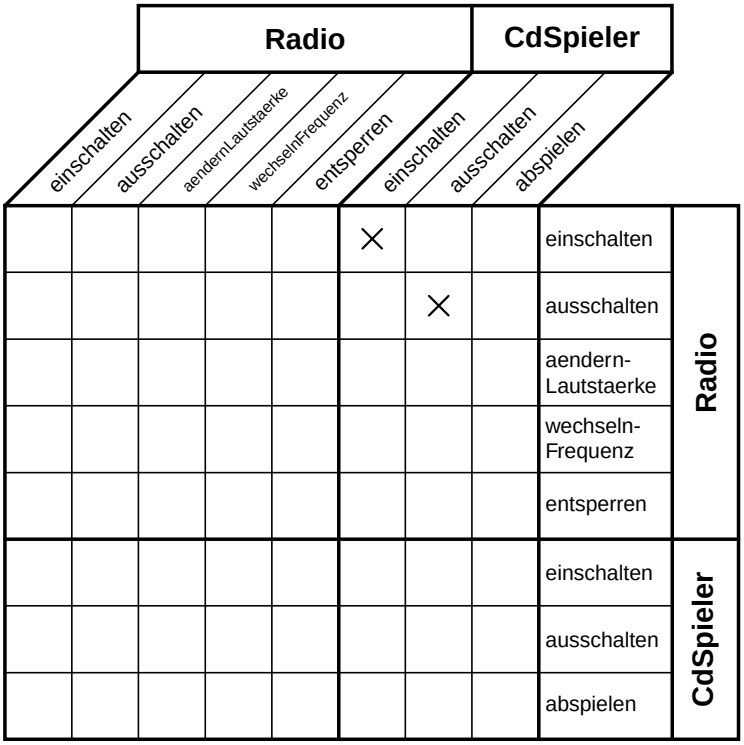


Abb. 145 Kontrollflußmatrix für die **Radio**-Chassis-Komponente

8.3 Entwurfsmodell VME

8.3.1 Die Nachteile eines konventionellen Entwurfs

Beim Übergang vom Analyse- zum Entwurfsmodell stellt die Umsetzung der im Analysemodell identifizierten Sequenzdiagramme einen zentralen Punkt dar. Ein Sequenzdiagramm hat dort für einen Anwendungsfall konfigurationsabhängig den Kontrollfluß zwischen dem Chassis- und seinen Komponentenobjekten beschrieben. Der konventionelle Ansatz für einen Entwurf wird typischerweise die vollständige Semantik des Anwendungsfalls in der zugehörigen Primär- bzw. Sekundäroperation abbilden. Innerhalb einer Operation wird jeweils abgefragt, welche Komponenten gerade installiert sind, um dann die Operationen der Komponentenobjekte auszuführen, die in dem entsprechenden Sequenzdiagramm spezifiziert sind. Dieser Ansatz besitzt mehrere Nachteile:

- Bei jedem Operationsaufruf wird die aktuelle Konfiguration ermittelt, was beim Vorliegen vieler Komponentenklassen signifikant Zeit kosten kann.
- Die Struktur der Operationen kann durch die erforderlichen Fallunterscheidungen sehr unübersichtlich werden.
- Die Aufrufbeziehungen zwischen den beteiligten Objekten sind oft sehr komplex und damit nur schwer nachvollziehbar.
- Soll eine bisher nicht verwendete Komponentenklasse in den Entwurf integriert werden, so können von den notwendigen Anpassungen alle Primär- und Sekundäroperationen der Chassis-Klasse betroffen sein.

Der Grund für das Auftreten dieser Nachteile liegt in der ***Verstreuung*** des ***Konfigurationswissens*** über die Primär- und Sekundäroperationen. Unter dem Begriff Konfigurationswissen soll hier nicht nur die aktuelle Konfiguration verstanden werden, sondern auch das Wissen, bei welcher Konfiguration welcher Kontrollfluß zu verwenden ist.

Im nächsten Abschnitt wird ein allgemeines Entwurfskonzept für Variantenmodelle vorgestellt, welches ***minimale Kontrollflüsse*** verwendet und das Konfigurationswissen ***zentral*** verwaltet.

8.3.2 Das statische Modell für zwei Hierarchieebenen

Den Ausgangspunkt für das statische Entwurfsmodell bilden die im ersten Schritt des statischen Analysemodells identifizierten Chassis- und Komponentenklassen, welche in Abb. 132 (S. 251) dargestellt sind. Da in der Analysephase nur alle öffentlichen Operationen spezifiziert werden, sind die Klassen um alle internen Operationen zu ergänzen. Die entstehenden Klassen werden als die ***korrespondierenden*** Klassen der Chassis- und Komponentenklassen bezeichnet und realisieren die Funktionalität. Für die Benennung einer korrespondierenden Klasse soll die Konvention gelten, daß ihr Name durch das Anhängen des Postfixes **Dsc**[3] an den entsprechenden Namen der Chassis- oder Komponentenklasse entsteht. Eine korrespondierende Klasse hat die Eigenschaft, daß sie keine Assoziationen zu anderen korrespondierenden Klassen besitzt. Daher sind ihre Operationen auch nicht in der Lage, Operationen anderer korrespondierender Klassen aufzurufen. Aus den korrespondierenden Klassen werden im zweiten

[3] Dsc steht als Abkürzung für Description.

Schritt die Chassis- und Komponentenklassen des Entwurfsmodells erzeugt.

Eine Komponentenklasse **X** besitzt die folgende Struktur:

- **X** erweitert die abstrakte Klasse **BaseComponent**.
- **X** besitzt eine Kompositionsbeziehung zu ihrer korrespondierenden Klasse.
- Für jeden öffentlichen Konstruktor aus der korrespondierenden Klasse **XDsc** wird ein öffentlicher Konstruktor erzeugt, der dieselben Parameter besitzt. Jeder Konstruktor hat die Aufgabe, das korrespondierende Objekt zu erzeugen.
- Falls die korrespondierende Klasse keinen öffentlichen Konstruktor bereitstellt, erhält **X** einen parameterlosen, öffentlichen Konstruktor, der das korrespondierende Objekt erzeugt.
- Zu jeder öffentlichen Operation aus **XDsc** existiert eine paketinterne Operation in **X** mit derselben Signatur und demselben Ergebnistyp.
- Die `delete`-Operation aus **BaseComponent** wird redefiniert.

In Abb. 146 ist die Struktur der Komponentenklasse **Radio** zusammen mit ihrer korrespondierenden Klasse dargestellt.

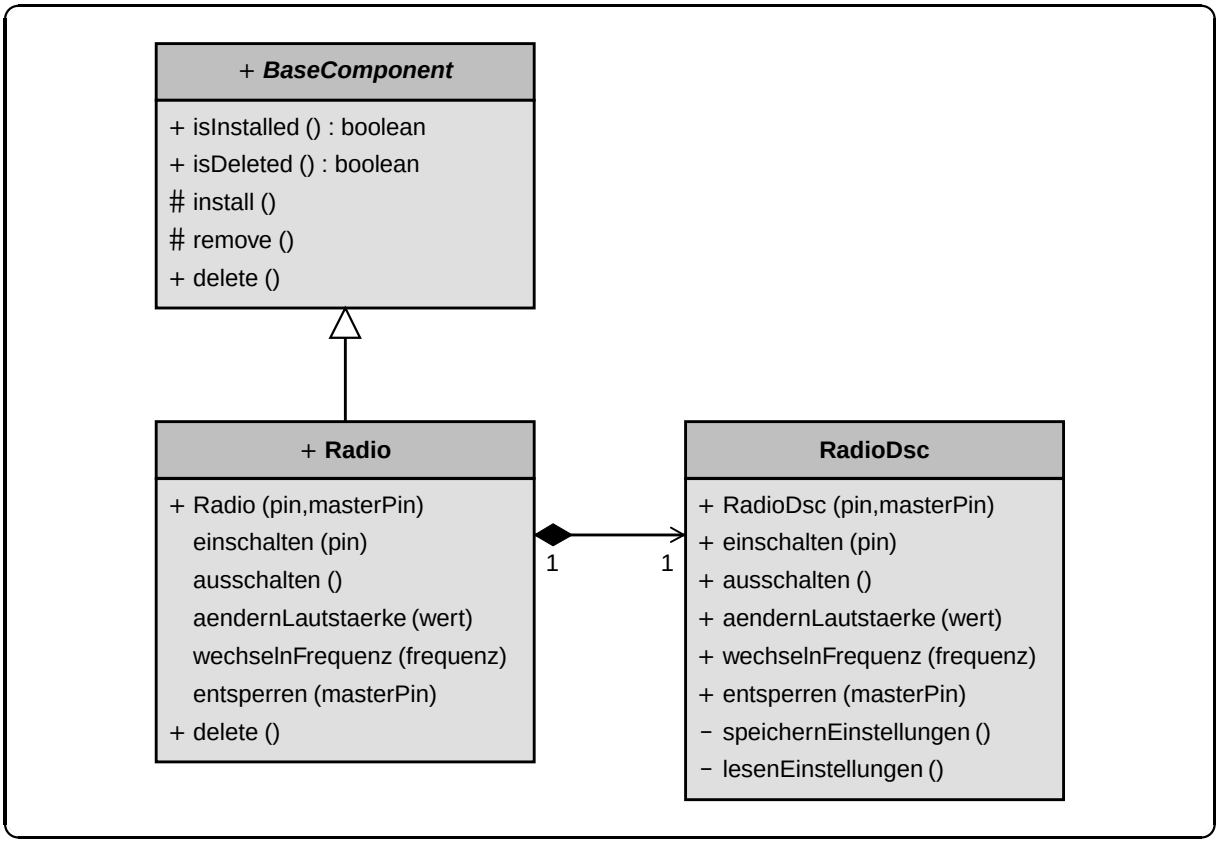


Abb. 146 Die Struktur der Komponentenklasse **Radio**

Der in Abb. 139 (S. 259) spezifizierte globale Zustandsautomat muß bei der Verwendung von JAVA als Implementierungssprache leicht verändert werden. Auf den entstehenden Zustandsautomaten wird in Abschnitt 8.3.4 (S. 273) näher eingegangen. Die Klasse **BaseComponent** realisiert den wesentlichen Teil dieses Zustandsautomaten. Nur die Kontrolle, ob eine Primäroperation ausgeführt werden darf, wird von der Komponentenklasse selbst übernommen.

Eine Chassis-Klasse **Y** ist wie folgt aufgebaut, wobei die Komponentenklassen $\mathbf{C}_1$ bis $\mathbf{C}_n$ zusammen mit ihren korrespondierenden Klassen $\mathbf{CDsc}_1$ bis $\mathbf{CDsc}_n$ vorliegen sollen:

- **Y** erweitert die abstrakte Klasse **BaseChassis**.
- **Y** besitzt eine Kompositionsbeziehung zu ihrer korrespondierenden Klasse.
- Außerdem liegen Kompositionsbeziehungen zu den Komponentenklassen $\mathbf{C}_1$ bis $\mathbf{C}_n$ vor.
- Für jeden öffentlichen Konstruktor aus der korrespondierenden Klasse **YDsc** wird ein öffentlicher Konstruktor generiert, der dieselben Parameter besitzt und das korrespondierende Objekt erzeugt.
- Falls **YDsc** keinen öffentlichen Konstruktor zur Verfügung stellt, wird für **Y** ein parameterloser, öffentlicher Konstruktor generiert, der das korrespondierende Objekt erzeugt.
- Zu jeder öffentlichen Operation aus der korrespondierenden Klasse **YDsc** existiert eine öffentliche Operation mit derselben Signatur und demselben Ergebnistyp.
- Aus den korrespondierenden Klassen $\mathbf{CDsc}_i$ werden diejenigen öffentlichen Operationen übernommen, die gemäß der Anforderung AV4 (vgl. S. 222) in **Y** sichtbar sein sollen.
- Für jede Komponentenklasse $\mathbf{C}_i$ werden die Konfigurationsoperationen $\texttt{installC}_i\texttt{(C}_i\texttt{)}$ und $\texttt{removeC}_i$ erzeugt. Falls $\texttt{k}>1$ Komponentenobjekte desselben Typs erlaubt sind, besitzen die Operationen als Parameter einen Indexwert mit dem Wertebereich $\{\texttt{0},\ldots,\texttt{k-1}\}$.
- Für jede Komponentenklasse $\mathbf{C}_i$ ist in **Y** die Verwaltungsoperation $\texttt{hasC}_i$ vorhanden. Sie besitzt als Parameter wieder einen Indexwert, wenn von einer Komponentenklasse mehrere Objekte vorliegen können.
- Die `delete`-Operation aus **BaseChassis** wird redefiniert.

Abb. 147 (S. 269) zeigt den Aufbau der Chassis-Klasse **Auto**. Die abstrakte Klasse **BaseChassis** hat die Aufgabe, einen einfachen Zustandsautomaten zu realisieren. Dieser wird im Abschnitt 8.3.4 (S. 273) vorgestellt.

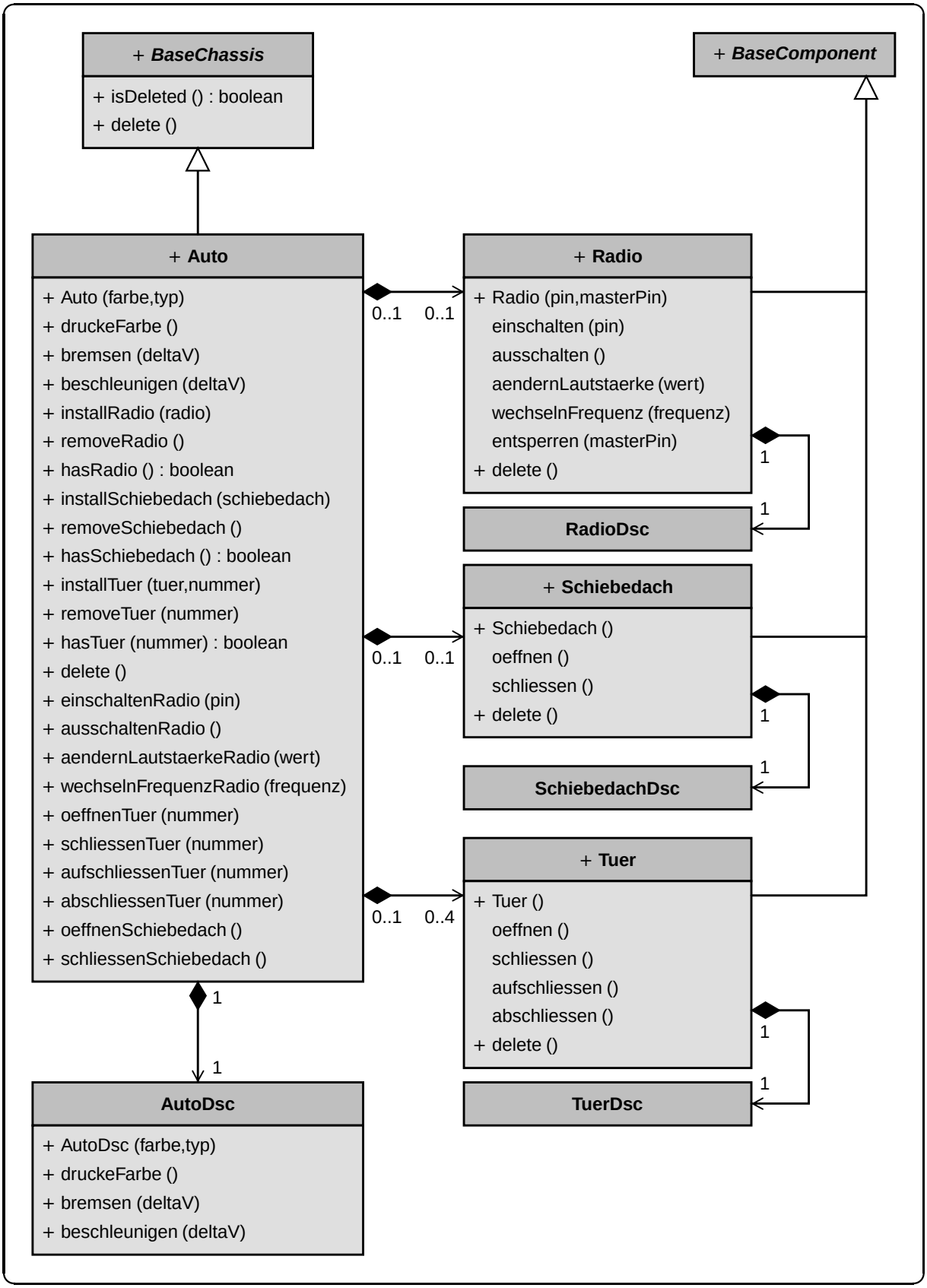


Abb. 147 Die Struktur der Chassis-Klasse **Auto**

8.3.3 Das statische Modell für mehr als zwei Hierarchieebenen

Neben den Chassis- und Komponentenklassen liegen jetzt auch Chassis-Komponentenklassen vor. Eine Chassis-Komponentenklasse **Z** übernimmt große Teile der Struktur einer Chassis-Klasse (vgl. S. 268). Für die Strukturbeschreibung von **Z** wird davon ausgegangen, daß als Unterkomponenten die Komponentenklassen $\mathbf{C}_1$ bis $\mathbf{C}_n$ und die Chassis-Komponentenklassen $\mathbf{CC}_1$ bis $\mathbf{CC}_k$ vorliegen mit ihren korrespondierenden Klassen $\mathbf{CDsc}_1$ bis $\mathbf{CDsc}_n$ und $\mathbf{CCDsc}_1$ bis $\mathbf{CCDsc}_k$. Mit diesen Definitionen läßt sich die Struktur von **Z** wie folgt spezifizieren:

- **Z** erweitert die abstrakte Klasse **BaseChassisComponent.**
  Diese Klasse besitzt zunächst dieselben Operationen wie **BaseComponent** (vgl. Abb. 146 (S. 267) und Abb. 148 (S. 271)). Sie wird aber trotzdem eingeführt, damit bereits anhand der Basisklasse zwischen Komponenten- und Chassis-Komponentenklassen unterschieden werden kann. Außerdem werden im Verlauf der Modellierung des dynamischen Modells weitere, unterschiedliche Operationen zu den Klassen **BaseComponent** und **BaseChassisComponent** hinzugefügt, so daß spätestens dann die Verwendung von zwei Klassen sinnvoll ist.
- Die Klasse **Z** besitzt eine Kompositionsbeziehung zu ihrer korrespondierenden Klasse **ZDsc**.
- Zusätzlich liegen Kompositionsbeziehungen zu allen Komponentenklassen $\mathbf{C}_i$ und Chassis-Komponentenklassen $\mathbf{CC}_j$ vor.
- Für jeden öffentlichen Konstruktor aus der korrespondierenden Klasse **ZDsc** wird ein öffentlicher Konstruktor in **Z** generiert, der diesselben Parameter besitzt und das korrespondierende Objekt erzeugt.
- Falls **ZDsc** keinen öffentlichen Konstruktor bereitstellt, wird für **Z** ein parameterloser, öffentlicher Konstruktor generiert, der das korrespondierende Objekt erzeugt.
- Zu jeder öffentlichen Operation aus **ZDsc** existiert in **Z** eine paketinterne Operation mit derselben Signatur und demselben Ergebnistyp.
- Aus den korrespondierenden Klassen $\mathbf{CDsc}_i$ werden diejenigen Operationen übernommen, die gemäß der Anforderung AV4 (vgl. S. 222) an der Schnittstelle von **Z** sichtbar sein sollen.
- Von den Chassis-Komponentenklassen $\mathbf{CC}_j$ werden diejenigen Primär- und Sekundäroperationen übernommen, die der Anforderung AV4 genügen.
- Für jede Komponentenklasse $\mathbf{C}_i$ werden die Konfigurationsoperationen $\texttt{installC}_i\texttt{(C}_i\texttt{)}$ und $\texttt{removeC}_i$ erzeugt. Falls $\texttt{m>1}$ Komponentenobjekte desselben Typs erlaubt sind, besitzen die Operationen als Parameter einen Indexwert mit dem Wertebereich $\{\texttt{0,...,m-1}\}$.
- Für jede Chassis-Komponentenklasse $\mathbf{CC}_j$ ist es notwendig, die Konfigurationsoperationen $\texttt{installCC}_j\texttt{(CC}_j\texttt{)}$ und $\texttt{removeCC}_j$ zu erzeugen. Falls $\texttt{m>1}$ Chassis-Komponentenobjekte desselben Typs erlaubt sind, besitzen die Operationen als Parameter einen Indexwert mit dem Wertebereich $\{\texttt{0,...,m-1}\}$.
- Für jede Komponentenklasse $\mathbf{C}_i$ gibt es in **Z** die Verwaltungsoperation $\texttt{hasC}_i$. Sie besitzt als Parameter wieder einen Indexwert, wenn von einer Komponentenklasse mehrere Objekte vorhanden sein können.
- Für jede Chassis-Komponentenklasse $\mathbf{CC}_j$ liegt in **Z** die Verwaltungsoperation $\texttt{hasCC}_j$ vor. Sie besitzt als Parameter wieder einen Indexwert, wenn von einer Chassis-Komponentenklasse mehrere Objekte existieren können.

- Die `delete`-Operation aus der **BaseChassisComponent**-Klasse wird redefiniert.

Die Struktur der Chassis-Komponentenklasse **Radio** ist in Abb. 148 zu sehen.

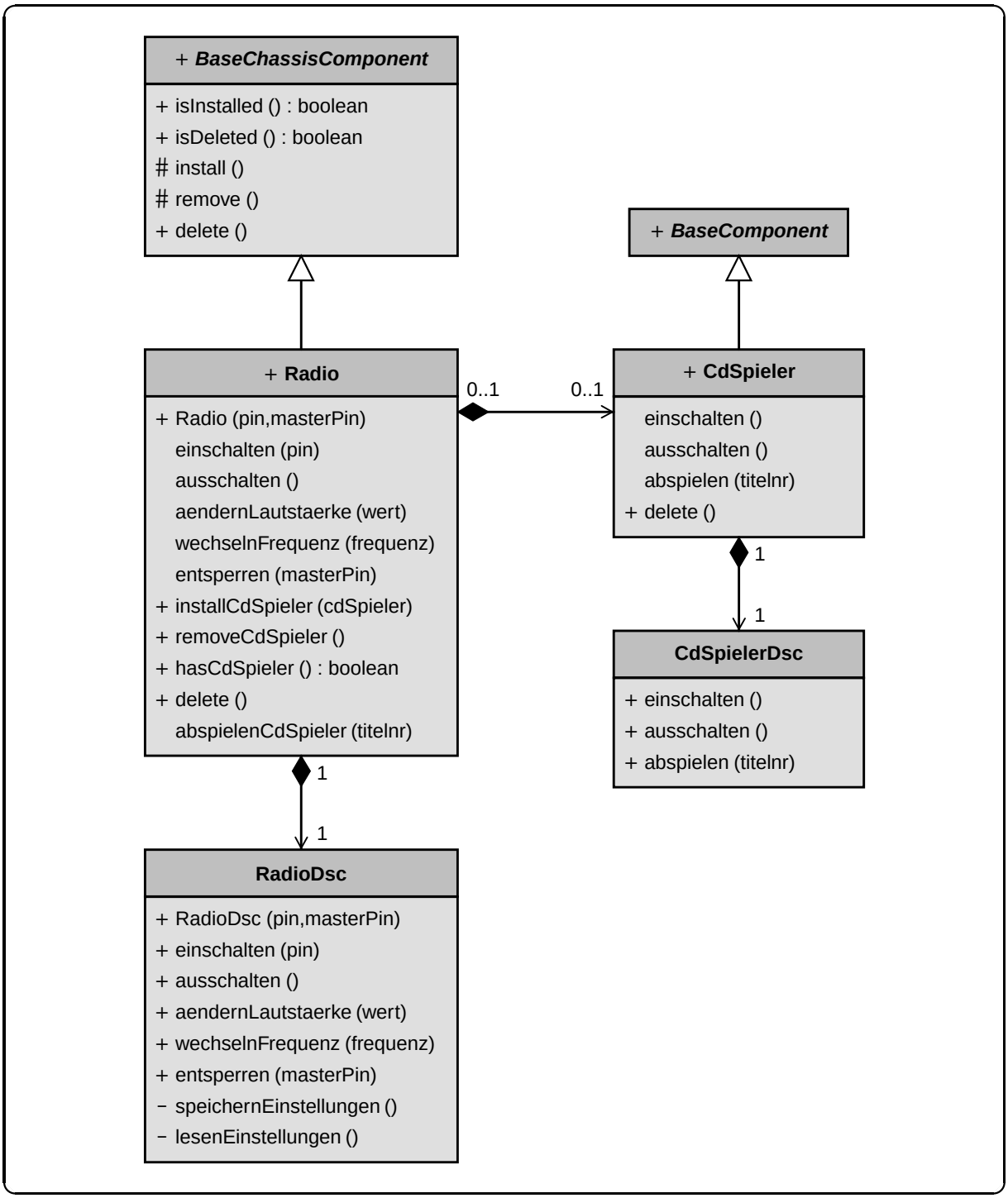


Abb. 148 Die Struktur der Chassis-Komponentenklasse **Radio**

Der Aufbau der Komponentenklassen ist von der Einführung der Chassis-Komponentenklassen nicht betroffen. Bei den Chassis-Klassen ist jetzt zu berücksichtigen, daß neben den Komponentenklassen auch Chassis-Komponentenklassen zur Definition der Unterkomponenten ver-

wendet werden können. Entsprechend sind bei der Strukturbeschreibung einer Chassis-Klasse die folgenden Ergänzungen im Vergleich zu der Beschreibung von S. 268 vorzunehmen, wobei die Chassis-Komponentenklassen $\mathbf{CC}_1$ bis $\mathbf{CC}_k$ vorliegen sollen:

- Zu allen Chassis-Komponentenklassen existieren Kompositionsbeziehungen.
- Von den Chassis-Komponentenklassen $\mathbf{CC}_j$ sind alle Primär- und Sekundäroperationen zu übernehmen, die der Anforderung AV4 genügen.
- Für alle $\mathbf{CC}_j$-Klassen sind $\texttt{installCC}_j\texttt{(CC}_j\texttt{)}$ und $\texttt{removeCC}_j$ als Konfigurationsoperationen zu erzeugen. Falls $\texttt{m}>1$ Chassis-Komponentenobjekte desselben Typs erlaubt sind, besitzen die Operationen als Parameter einen Indexwert mit dem Wertebereich $\{\texttt{0,...,m-1}\}$.
- Für jede Chassis-Komponentenklasse $\mathbf{CC}_j$ ist die Verwaltungsoperation $\texttt{hasCC}_j$ vorhanden. Sie besitzt als Parameter wieder einen Indexwert, wenn von einer Chassis-Komponentenklasse mehrere Objekte vorliegen können.

8.3.4 Das dynamische Modell für zwei Hierarchieebenen

8.3.4.1 Globale Zustandsautomaten

Der Zustandsautomat für Komponentenobjekte aus Abb. 139 (S. 259) muß bei einer JAVA-Realisierung berücksichtigen, daß in JAVA keine Destruktoren zur Verfügung stehen. Daher wird in allen Zuständen eine `isDeleted`-Operation zugelassen, die die Information liefert, ob sich das Objekt im *Deleted*-Zustand befindet. Wenn die letzte Referenz auf das Objekt gelöscht wird, kann der zugehörige Speicherplatz wieder vom JAVA Garbage Collector freigegeben werden. Für Chassis-Objekte wird ebenfalls ein globaler Zustandsautomat definiert, der nur die beiden Zustände *Existing* und *Deleted* kennt. Die beiden Zustandsautomaten sind in Abb. 149 und Abb. 150 dargestellt.

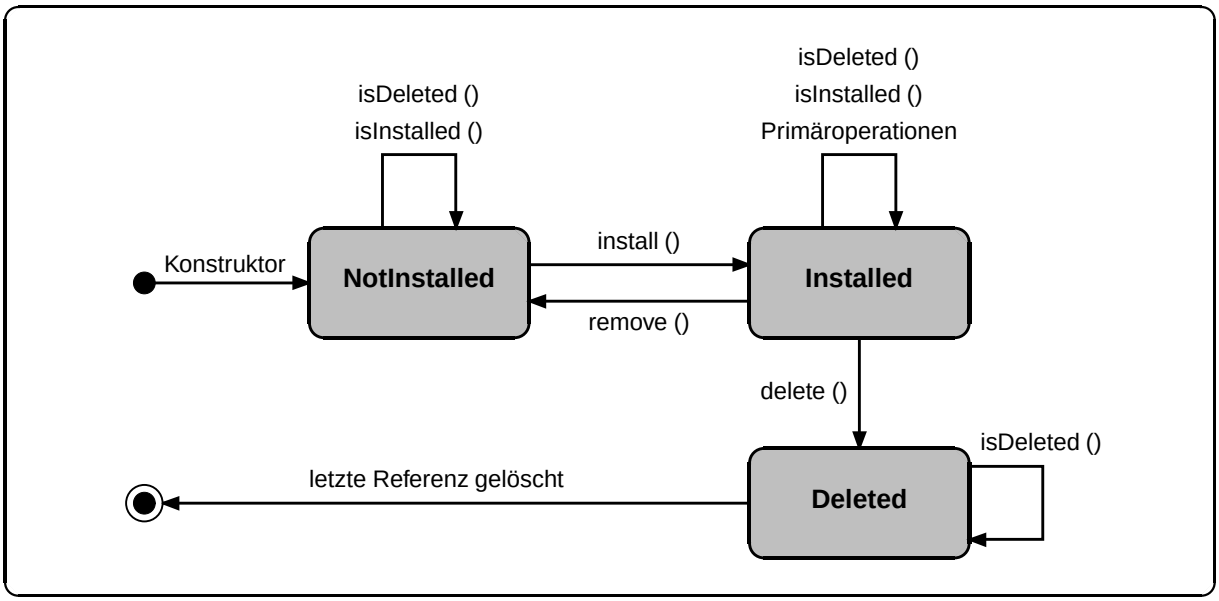


Abb. 149 Der globale Zustandsautomat einer Komponentenklasse

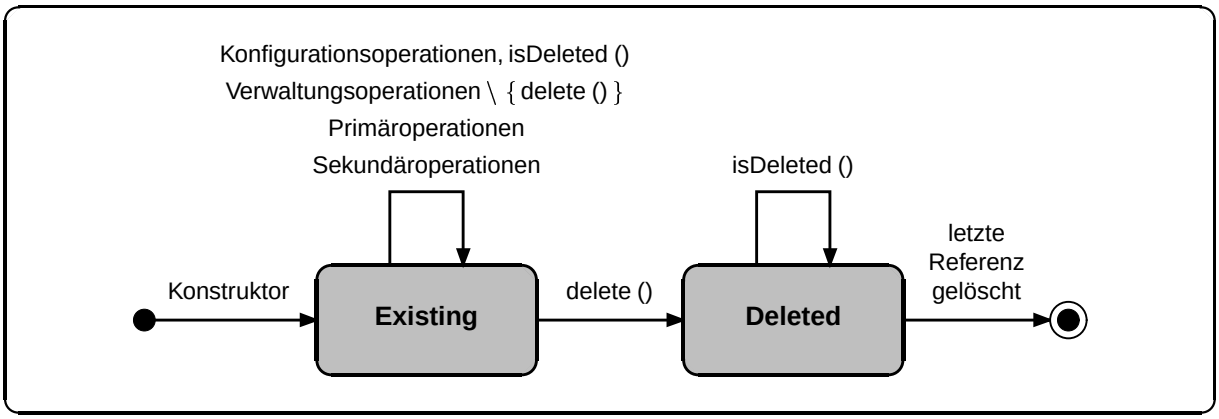


Abb. 150 Der globale Zustandsautomat einer Chassis-Klasse

Wenn in der Analysephase zusätzliche lokale Zustandsautomaten spezifiziert wurden, dann verfeinern diese die Zustände *Existing* bzw. *Installed*.

8.3.4.2 Die Umsetzung der Sequenzdiagramme

Um eine direkte Umsetzung der in der Analysephase identifizierten Sequenzdiagramme zu erreichen, wird für jedes Sequenzdiagramm eine eigene Klasse erstellt. Ein Objekt dieser Klasse erhält bei seiner Erzeugung die Informationen, welche Objekte von dem Sequenzdiagramm betroffen sind. Derartige Klassen werden in der Folge als ***Script-Klassen*** bezeichnet. Eine Script-Klasse besitzt in der Regel als einzige Operation die Primär- oder Sekundäroperation, für die das entsprechende Sequenzdiagramm definiert ist. Die Ausnahme bilden überladene Operationen. Um eine automatische Generierung von Script-Klassen zu vereinfachen, werden alle überladenen Operationen in einer Script-Klasse abgebildet. Damit werden bei n überladenen Operationen auch n Sequenzdiagramme über diese Script-Klasse realisiert. Falls dieses nicht gewünscht ist, können einfach andere Operationsnamen gewählt werden, um ein Überladen zu vermeiden. Intern realisiert das Script-Objekt dann genau die über das Sequenzdiagramm spezifizierten Aufrufbeziehungen zwischen den beteiligten Objekten.

In Abb. 151 ist die Struktur der Klasse **AutoTuerAbschliessenRadioSchiebedach** dargestellt, die das Sequenzdiagramm aus Abb. 141 (S. 260) umsetzt.

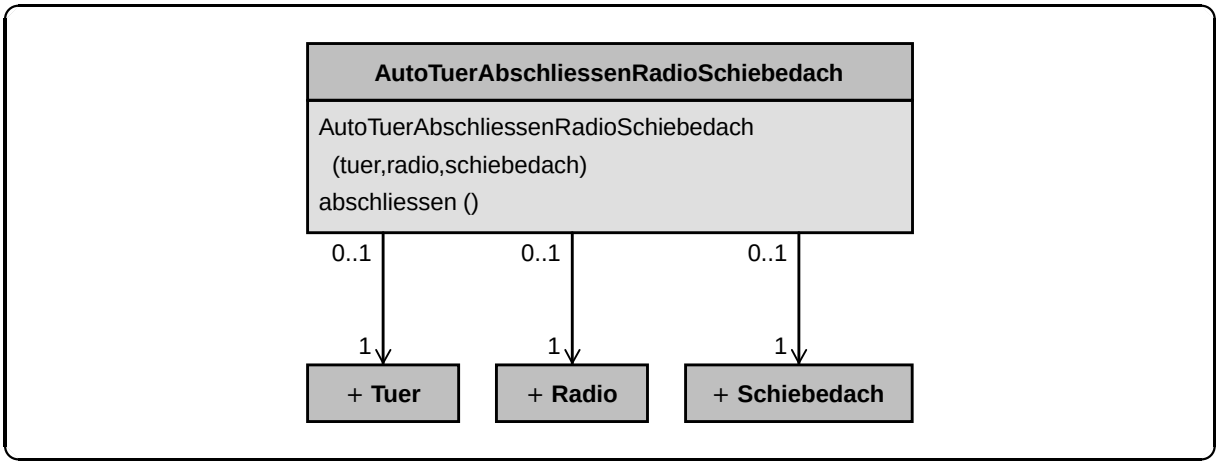


Abb. 151 Die Script-Klasse **AutoTuerAbschliessenRadioSchiebedach**

Für die Namensgebung der Script-Klassen gilt die folgende Konvention:

- Der Name beginnt mit dem Namen der Chassis-Klasse (hier: **Auto**).
- Wenn eine Sekundäroperation vorliegt, folgt der Name der Komponentenklasse, zu der diese Operation gehört (hier: **Tuer**).
- Der nächste Bestandteil ist der Operationsname, dessen erster Buchstabe groß geschrieben wird (hier: **Abschliessen**).
- Den Abschluß bilden die weiteren Komponentenklassen, die im Sequenzdiagramm aufgeführt sind (hier: **Radio** und **Schiebedach**).

Mit diesen Regeln ist sichergestellt, daß für alle vorliegenden Sequenzdiagramme eindeutige Klassennamen erzeugt werden, wenn man berücksichtigt, daß die Sequenzdiagramme für überladene Operationen auf diesselbe Script-Klasse abgebildet werden.

Neben den Script-Klassen werden zusätzlich für alle Primäroperationen der Chassis-Klassen und der Komponentenklassen Interface-Klassen erzeugt. Diese bilden später die Grundlage für einen Austausch von Script-Objekten zur Laufzeit, um auf Strukturänderungen eines Chassis-Objekts, d.h. den Ein- und Ausbau von Komponentenobjekten, zu reagieren. Die Script-Klassen der Sequenzdiagramme, die zu derselben Primär- oder Sekundäroperation der Chassis-Klasse gehören, implementieren alle dasselbe Interface.

Wenn man für die Operation `abschliessen` der Komponentenklasse **Tuer** in Abhängigkeit einer Radio- und Schiebedachinstallation vier Sequenzdiagramme definiert, dann ergibt sich der in Abb. 152 gezeigte Zusammenhang zwischen dem Script-Interface und den Script-Klassen.

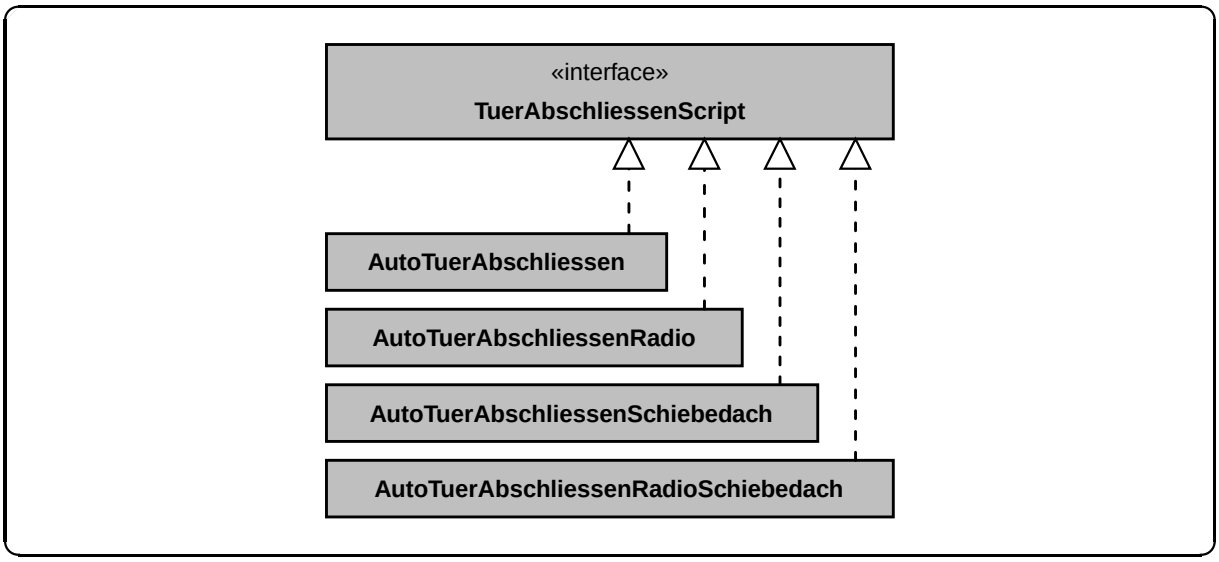


Abb. 152 Das Script-Interface **TuerAbschliessenScript**

Die Interface-Namen haben den folgenden Aufbau:

- Der erste Namensbestandteil ist der Name der Komponenten- oder Chassis-Klasse (hier: **Tuer**).
- Es folgt der Name der Operation, dessen erster Buchstabe groß geschrieben wird (hier: **Abschliessen**).
- An den Operationsnamen wird dann noch **Script** angehängt.

Wie bei den Script-Klassen gilt, daß für eine Menge von überladenen Operationen einer Chassis- oder Komponentenklasse nur ein Script-Interface erzeugt wird.

8.3.4.3 Die Erzeugung eines Chassis-Objekts

Während der Erzeugung eines Chassis-Objekts sind für die vorhandenen Primäroperationen geeignete Script-Objekte auszuwählen. Da zum Erzeugungszeitpunkt keine Komponentenobjekte vorliegen, werden für die Sekundäroperationen noch keine Script-Objekte benötigt.

Die für die Chassis-Objekterzeugung erforderliche Konfigurationsfunktionalität wird in einer ei-

genen Konfigurationsklasse bereitgestellt. Für das Autobeispiel erfüllt die Klasse **AutoConfiguration** diese Aufgabe (siehe Abb. 153).

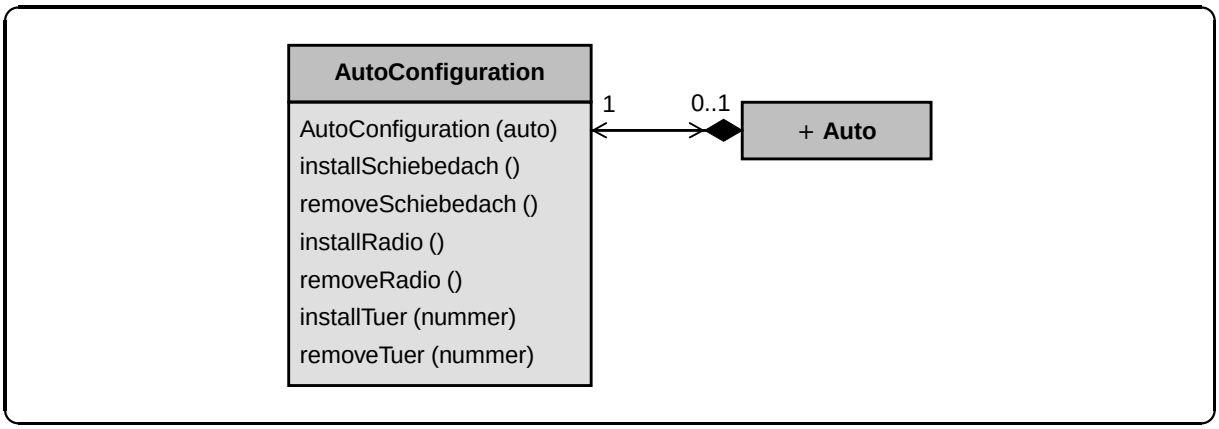


Abb. 153 Die Konfigurationsklasse für ein Autoobjekt

Insgesamt entsteht damit für das dynamische Modell das in Abb. 154 (S. 277) dargestellte Klassendiagramm. Mit der Erzeugung des Chassis-Objekts wird auch ein Konfigurationsobjekt angelegt. Der Konstruktor des Konfigurationsobjekts wählt die notwendigen Script-Klassen aus, erzeugt die Script-Objekte und installiert sie im Chassis-Objekt. Im Beispiel aus Abb. 153 sind dies die Script-Objekte der Klassen **AutoDruckeFarbeScript**, **AutoBremsenScript** und **AutoBeschleunigenScript**. Das entsprechende Sequenzdiagramm ist in Abb. 155 (S. 278) dargestellt. Damit können die Primäroperationen auf einem Chassis-Objekt ausgeführt werden.

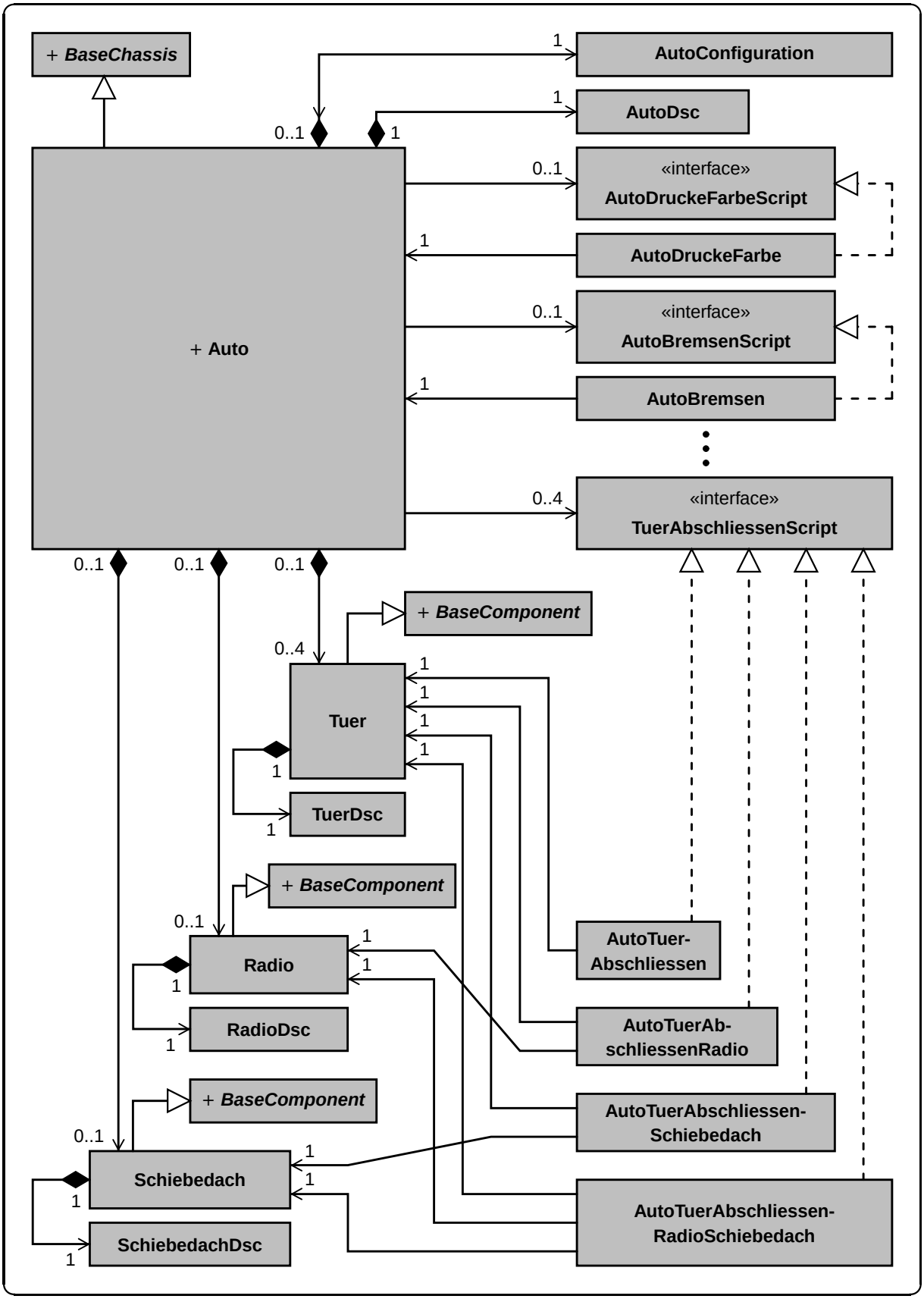


Abb. 154 Das Klassendiagramm des dynamischen Modells für das Autobeispiel

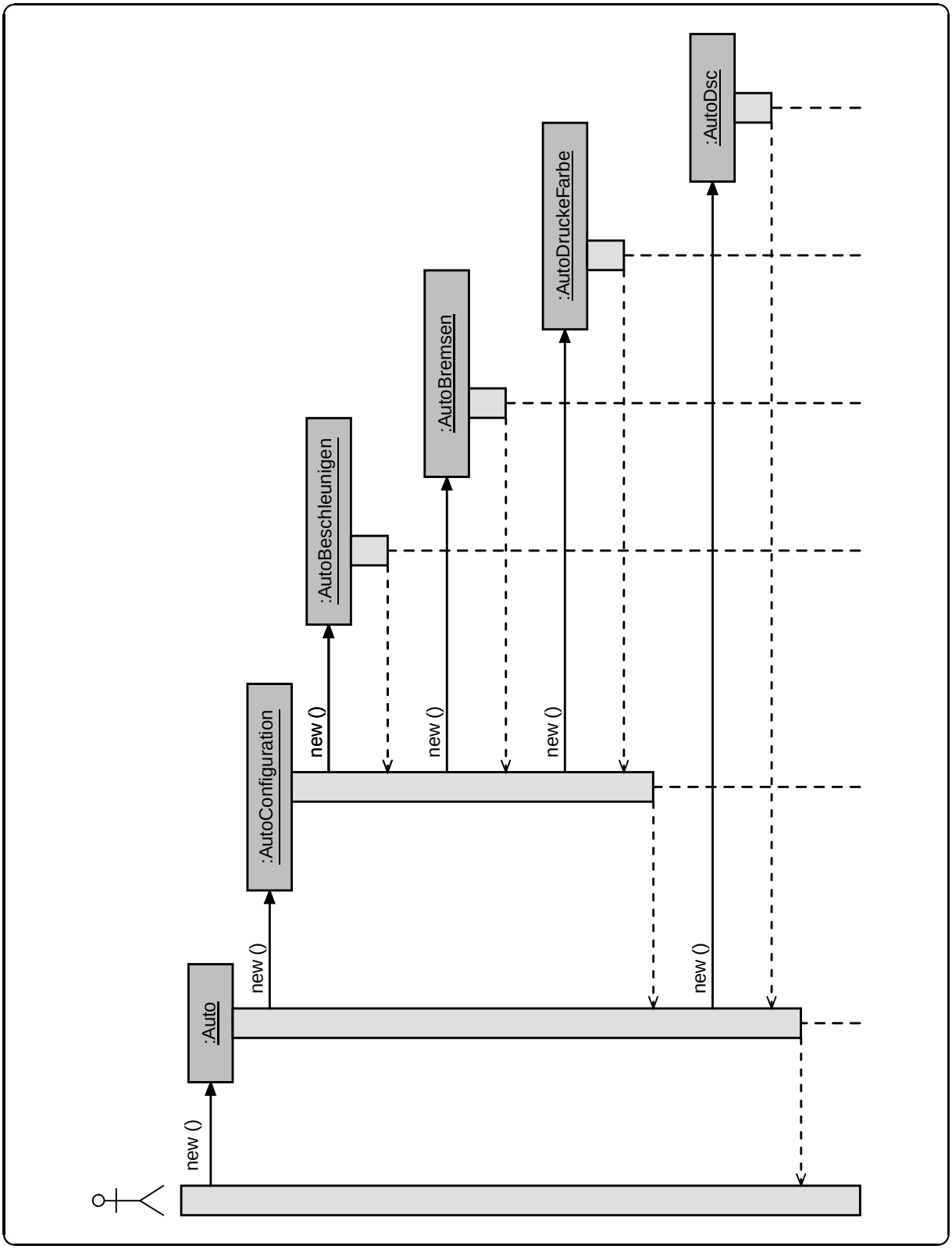


Abb. 155 Das Sequenzdiagramm für die Erzeugung eines Autoobjekts

8.3.4.4 Der Aufruf einer Primäroperation für ein Chassis-Objekt

Der Aufruf einer Primäroperation vollzieht sich in drei Schritten:

- Zunächst wird überprüft, ob sich das Chassis-Objekt nicht bereits im Zustand *Deleted* befindet.
- Der zweite Schritt kontrolliert, ob das notwendige Script-Objekt installiert ist.
- Als letztes wird die Operation des Script-Objekts aufgerufen, die den Kontrollfluß zwischen dem Chassis-Objekt und den Komponentenobjekten steuert.

Ist einer der ersten beiden Schritte nicht erfolgreich, wird der Aufruf der Primäroperation mit einer entsprechenden Exception beendet. Diese beiden Schritte werden von der `checkCall`-Operation ausgeführt, die der **BaseChassis**-Klasse zugeordnet ist. Abb. 156 enthält das Sequenzdiagramm der Operation `beschleunigen`[4]. Das Script-Objekt **AutoBeschleunigen** hat hier nur die Aufgabe, die entsprechende Operation auf dem korrespondierenden Objekt **AutoDsc** aufzurufen, da das Beschleunigen selbst keine Auswirkungen auf den Kontrollfluß anderer Komponentenobjekte besitzt.

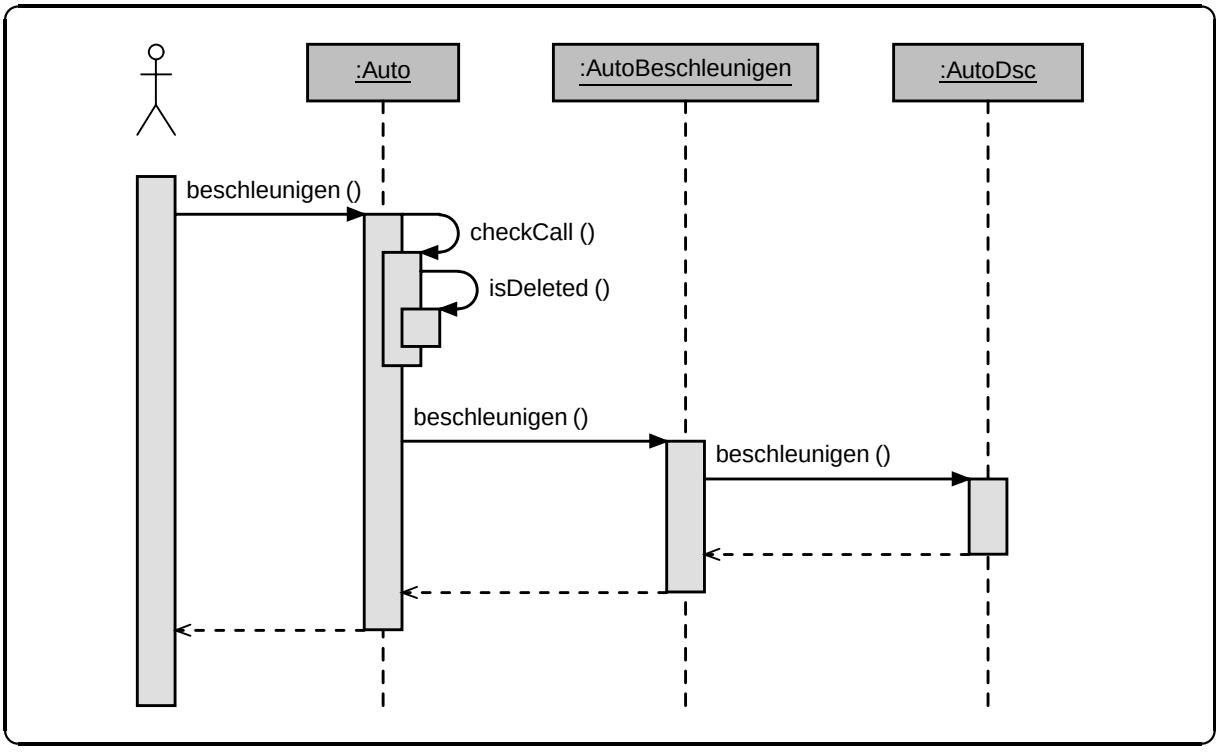


Abb. 156 Das Sequenzdiagramm für die Primäroperation `beschleunigen`

[4] Der `deltaV`-Parameter wurde weggelassen, da er für den globalen Kontrollfluß keine Rolle spielt.

8.3.4.5 Der Einbau einer Komponente

In der Chassis-Klasse existiert für jede einbaubare Komponente $\mathbf{C}_i$ eine `installC`$_i$-Operation. Die Operation führt zunächst einen Test aus, ob die Komponente überhaupt eingebaut werden darf. Hierfür müssen die folgenden Bedingungen gelten:

- Das Chassis-Objekt darf nicht gelöscht sein.
- Eine entsprechende Komponente darf nicht bereits im Chassis-Objekt existieren.
  Falls nur eine Komponente dieses Typs in ein Chassis-Objekt eingebaut werden darf, wird kontrolliert, daß nicht bereits eine derartige Komponente existiert. Sind mehrere Komponenten erlaubt, wird getestet, ob unter der in der `install`-Operation angegebenen Identifikation bereits eine Komponente vorhanden ist.
- Die neue Komponente darf nicht bereits gelöscht sein.
- Die neue Komponente darf nicht schon in einem anderen Chassis-Objekt eingebaut sein, d.h. sie muß sich im Zustand *NotInstalled* befinden.

Zur Überprüfung dieser Bedingungen wird in der **BaseChassis**-Klasse eine `checkInstall`-Operation bereitgestellt. War der Test erfolgreich, wird im Chassis-Objekt die Beziehung zu dem Komponentenobjekt hergestellt. Im nächsten Schritt ist zu überprüfen, ob sich durch den Einbau des Komponentenobjekts Kontrollflüsse verändern. Dies wäre beispielsweise beim Einbau eines Radios der Fall, wenn die Operation `abschliessen` der Klasse **Tuer** beim Vorhandensein eines Radios dieses ausschalten soll. Hierbei ist allerdings zu beachten, daß eine Kontrollflußänderung nur für die installierten **Tuer**-Objekte erforderlich ist (vgl. hierzu auch Abb. 142 (S. 261)).

Die Anpassung der Kontrollflüsse wird vom Konfigurationsobjekt übernommen. Hierfür stellt die Konfigurationsklasse für jede einzelne Komponentenklasse $\mathbf{C}_i$ eine `installC`$_i$-Operation zur Verfügung (vgl. Abb. 153 (S. 276)).

Die `installC`$_i$-Operation der Konfigurationsklasse wird von der entsprechenden `installC`$_i$-Operation der Chassis-Klasse aufgerufen. Sie analysiert die neue Konfiguration und tauscht über ihre Beziehung zur Chassis-Klasse dort die betroffenen Script-Objekte aus. Als letztes wird das Komponentenobjekt durch den Aufruf der `install`-Operation in den *Installed*-Zustand versetzt.

Der Kontrollfluß für das erfolgreiche Einbauen eines Radioobjekts in ein Autoobjekt ist in dem Sequenzdiagramm aus Abb. 157 (S. 281) beschrieben. Das Script-Objekt der Klasse **AutoTuerAbschliessenRadio** ersetzt dabei das bisher vorhandene Script-Objekt der Klasse **AutoTuerAbschliessen**. Bei einer konkreten Implementierung der Konfigurationsklasse ist dann festzulegen, ob der neue Kontrollfluß nur für die Fahrertür, die als `Tuer[0]` definiert werden könnte, oder für alle Türen zu installieren ist.

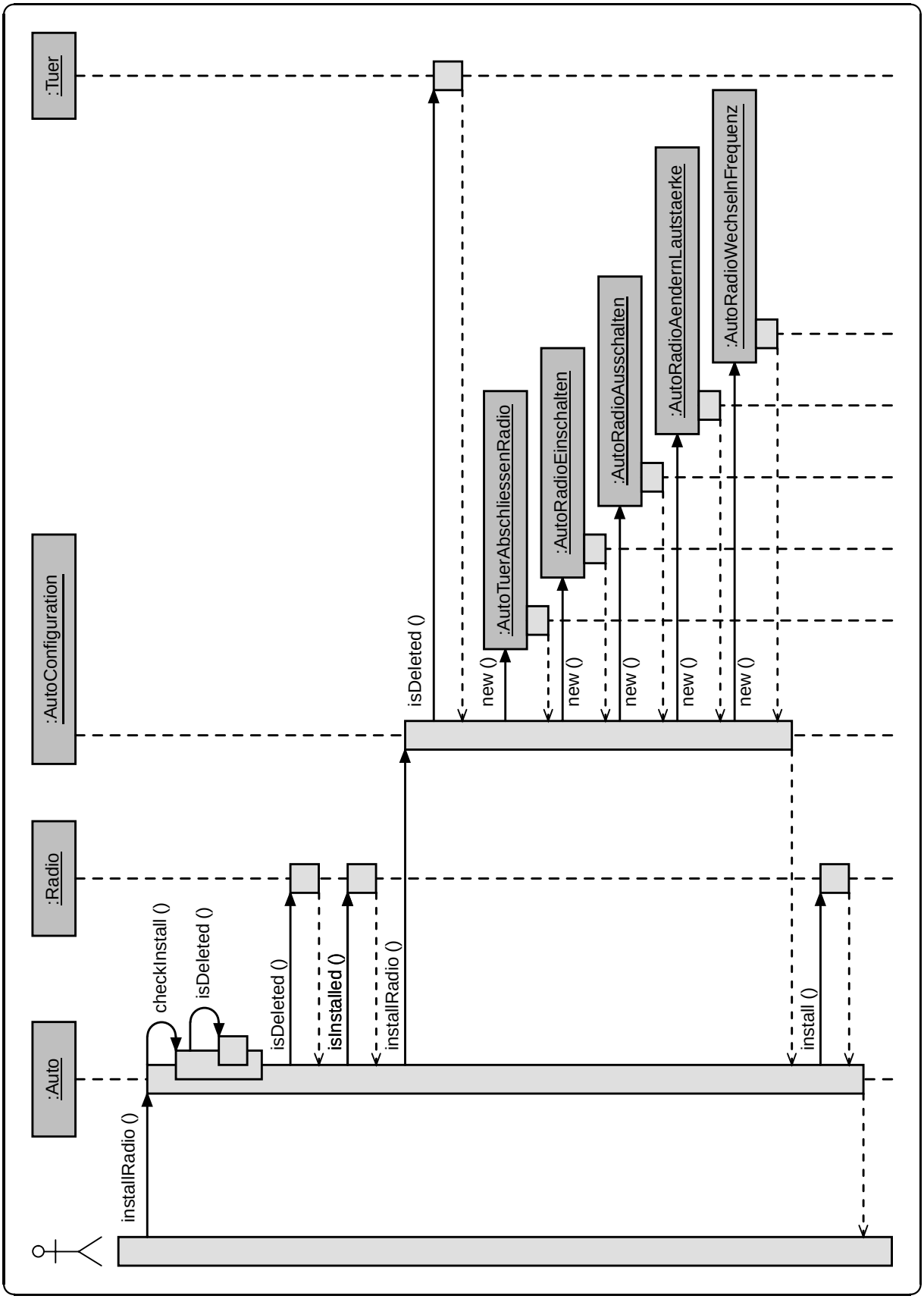


Abb. 157 Das Sequenzdiagramm für den Einbau eines Radios

8.3.4.6 Der Aufruf einer Sekundäroperation

Nach dem erfolgreichen Einbau einer Komponente sind alle über diese Komponente definierten Sekundäroperationen für das Chassis-Objekt aufrufbar. Die prinzipielle Bearbeitung eines Operationsaufrufs unterscheidet sich nicht von derjenigen einer Primäroperation (vgl. Abschnitt 8.3.4.4 (S. 279)). Nach der Aktivierung des Script-Objekts testet dieses zunächst, ob die benötigten Komponentenobjekte installiert sind und sich nicht im Zustand *Deleted* befinden. Danach werden die Operationen auf den Komponentenobjekten aufgerufen, die diese Aufrufe an die korrespondierenden Objekte weiterleiten.

Für das Autobeispiel ist in Abb. 158 (S. 283) das Sequenzdiagramm der `abschliessen-Tuer`-Operation dargestellt. Da ein Radio installiert ist, wird es zusätzlich zum Abschließen der Tür ausgeschaltet. Die Operation `abschliessen` der Script-Klasse **AutoTuerAbschliessenRadio** ist hier so aufgebaut, daß eine Zustandsänderung auf einem der beteiligten Objekte erst vorgenommen wird, nachdem für alle beteiligten Objekte die Tests erfolgreich durchgeführt wurden. Tritt dagegen ein Fehler auf, weil beispielsweise eine benötigte Komponente nicht installiert ist, wird eine entsprechende Exception erzeugt. Diese kann dann vom Script-Objekt selbst behandelt oder nach außen weitergereicht werden.

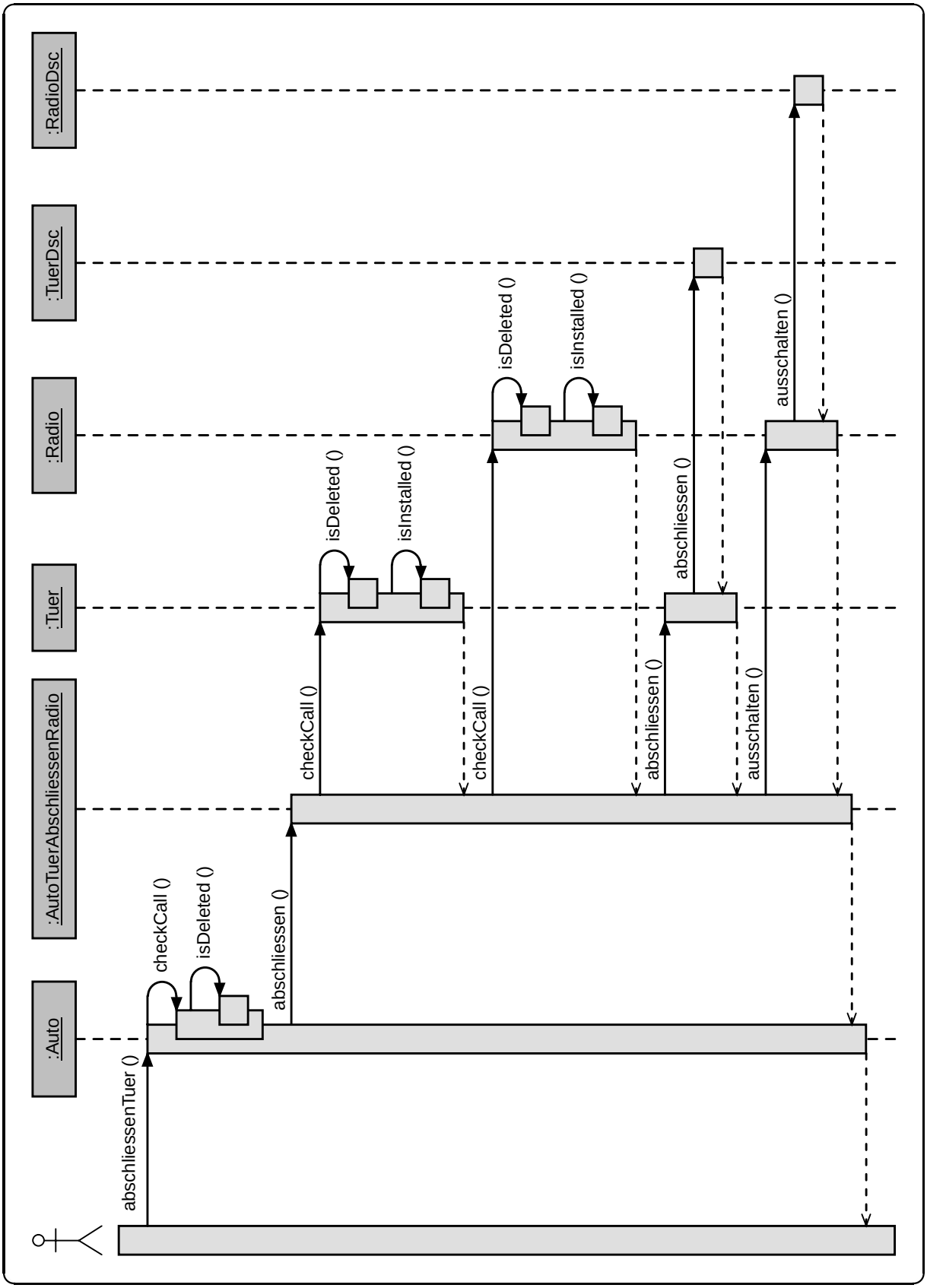


Abb. 158 Das Sequenzdiagramm für das Abschließen einer Tür

8.3.4.7 Der Ausbau einer Komponente

Die Chassis-Klasse besitzt für jede Komponentenklasse $\mathbf{C}_i$ des Variantenmodells eine `removeC`$_i$-Operation. Wenn mehrere Komponentenobjekte desselben Typs in ein Chassis-Objekt einbaubar sind, dann wird das zu entfernende Komponentenobjekt in der `removeC`$_i$-Operation über einen Parameter `nummer` identifiziert.

Für die Ausführung der `removeC`$_i$-Operation sind vier Schritte notwendig:

- Als erstes wird überprüft, ob sich das Chassis-Objekt im *Existing*-Zustand befindet.
- Der zweite Schritt kontrolliert, ob überhaupt ein entsprechendes Komponentenobjekt installiert ist.
- Im nächsten Schritt wird die `removeC`$_i$-Operation des Konfigurationsobjekts aufgerufen. Diese ermittelt anhand der vorliegenden Konfiguration, welche Script-Objekte zu ersetzen sind.
- Die letzte Aktion ist der Aufruf der `remove`-Operation auf dem Komponentenobjekt, um dieses in den Zustand *NotInstalled* zu versetzen.

Ob sich die Komponente im Zustand *Deleted* befindet, wird nicht überprüft, da es möglich sein muß, eine zerstörte Komponente zu entfernen.

Der Kontrollfluß für das Ausbauen des Radios ist in Abb. 159 beschrieben. Da das Abschließen

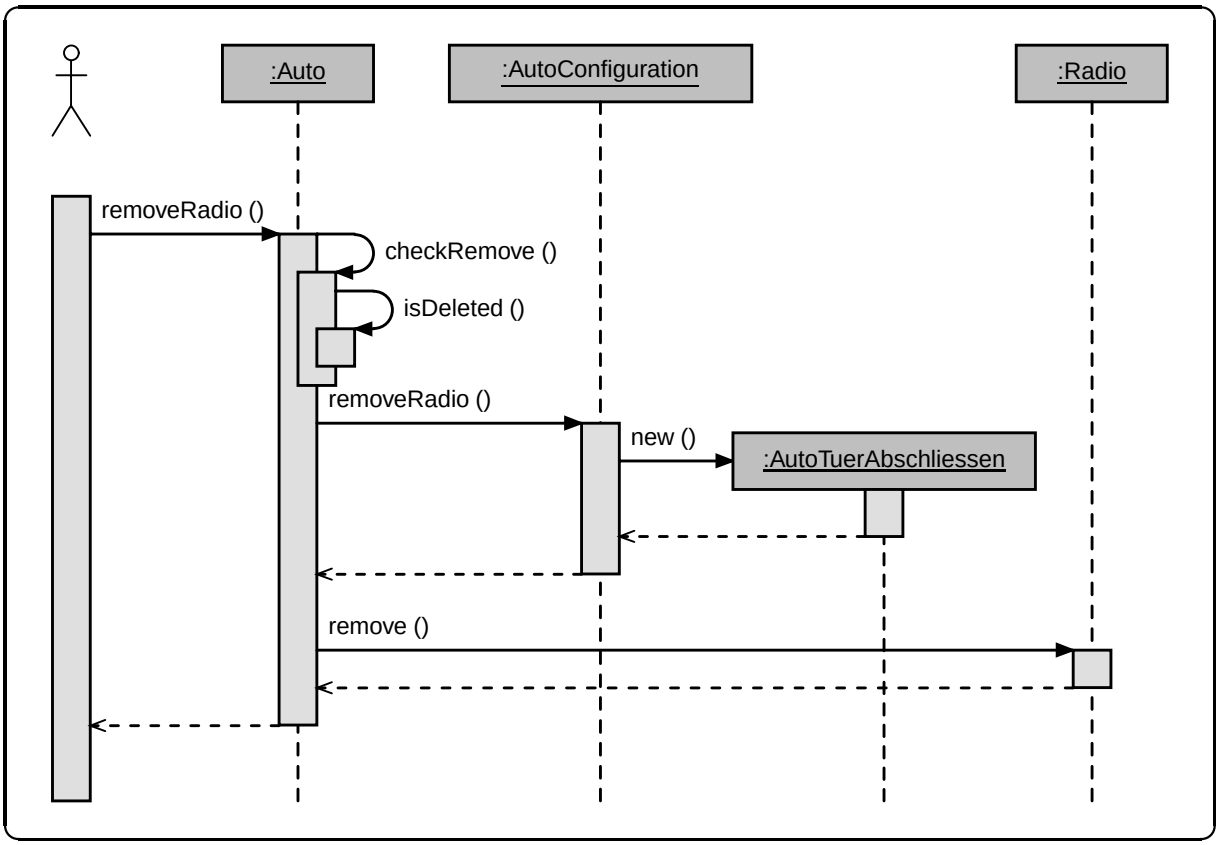


Abb. 159 Das Sequenzdiagramm für den Ausbau eines Radios

einer Tür auch das Radio betrifft, ist aufgrund des Radioausbaus das aktuelle Script-Objekt für

die Operation `abschliessenTuer` durch ein neues Script-Objekt der Klasse **AutoTuerAbschliessen** zu ersetzen. Hierbei ist beim Entwurf der Konfigurationsklasse wieder festzulegen, ob von diesem Austausch alle Türen betroffen sind, oder nur die Fahrertür.

8.3.4.8 Das Löschen von Chassis- und Komponentenobjekten

Der Aufruf der `delete`-Operation für ein Chassis-Objekt besitzt die folgende Semantik:

- Die Beziehung zu dem korrespondierenden Objekt wird gelöscht. Damit wird die einzige Referenz auf dieses Objekt vernichtet, wodurch es für den JAVA Garbage Collector freigegeben wird.
- Für jedes Komponentenobjekt wird die `delete`-Operation aufgerufen, sofern dieses sich nicht bereits im *Deleted*-Zustand befindet.
- Alle Beziehungen zu den Komponentenobjekten werden abgebaut.
- Das Chassis-Objekt wird in den Zustand *Deleted* versetzt.

Die `delete`-Operation für ein Komponentenobjekt hat eine sehr einfache Struktur:

- Die Beziehung zu dem Objekt der korrespondierenden Klasse wird abgebaut, um dieses für den JAVA Garbage Collector freizugeben.
- Das Komponentenobjekt wechselt in den Zustand *deleted*.

8.3.4.9 Die Struktur der Klassen BaseChassis und BaseComponent

Abb. 160 zeigt die Struktur der abstrakten Klassen **BaseChassis** und **BaseComponent**, die entsteht, wenn man die `check`-Operationen aus den letzten Abschnitten berücksichtigt und eine neue abstrakte Oberklasse **VariationModelObject** einführt, die für das logische Löschen eines Objekts zuständig ist.

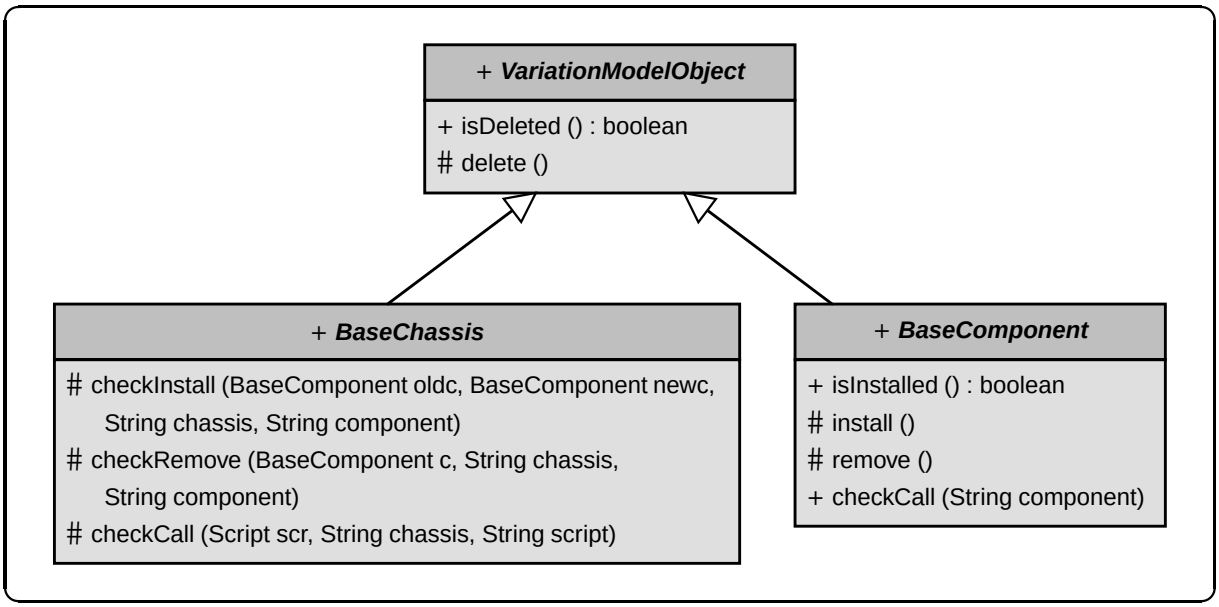


Abb. 160 Die Struktur der Klassen **BaseChassis** und **BaseComponent**

8.3.4.10 Die Struktur der Script-Klassen

Eine zentrale Anforderung an die Struktur eines Script-Objekts besteht darin, daß es nach der Operationsausführung ***zustandslos*** ist. Damit ist es möglich, zur Laufzeit Script-Objekte auszutauschen, um Kontrollflüsse anzupassen. Während der Ausführung der Operation eines Script-Objekts dürfen natürlich Zustandsinformationen aufgebaut werden, die für die Steuerung des Kontrollflusses zwischen den beteiligten Objekten notwendig sind.

Die einfachste Form eines Script-Objekts liegt vor, wenn beim Aufruf einer Primäroperation des Chassis-Objekts nur eine oder mehrere Operationen des korrespondierenden Chassis-Objekts zu aktivieren sind (vgl. Abb. 161).

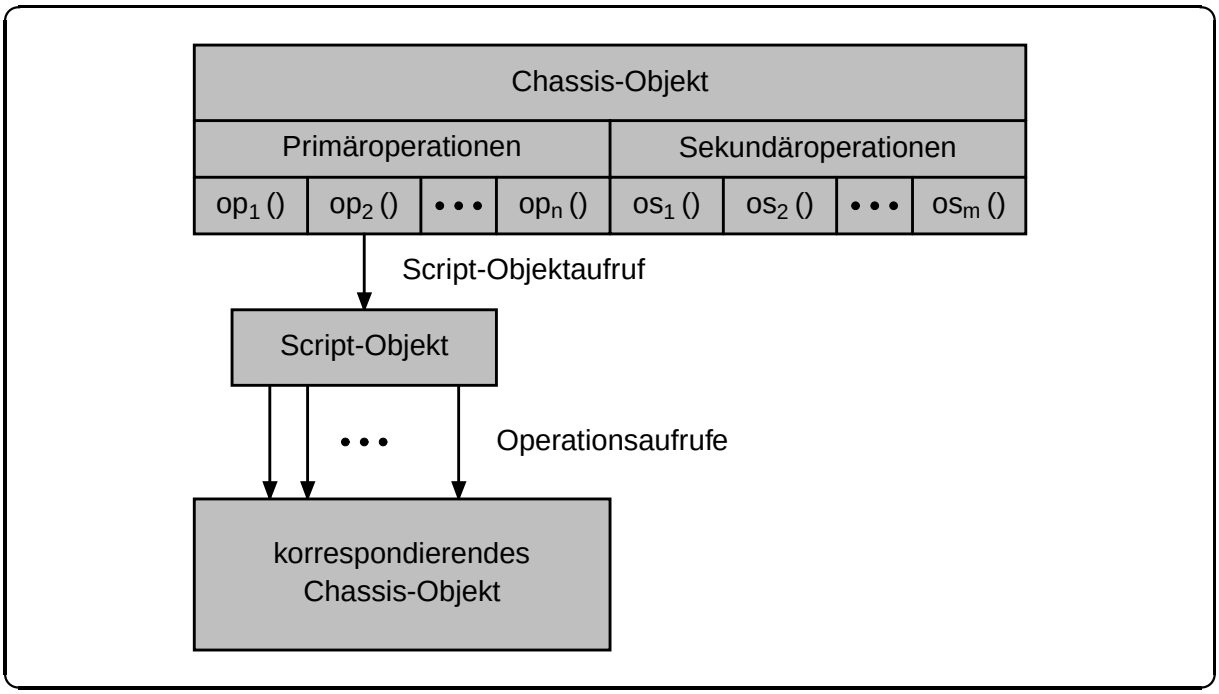


Abb. 161 Die einfachste Struktur eines Script-Objekts

Prinzipiell ist es auch möglich, daß der Aufruf einer Sekundäroperation letztlich nur das korrespondierende Chassis-Objekt betrifft. Dieser Fall sollte aber eher selten auftreten, da der Aufruf einer Sekundäroperation in der Regel auch zu einem Zugriff auf das zugehörige Komponentenobjekt führen wird.

Abb. 162 (S. 287) zeigt den Kontrollfluß beim Aufruf einer Primäroperation, die auch Komponentenobjektaufrufe zur Folge hat. Die Ausführung der Operationen auf den korrespondierenden Komponentenobjekten wird ausschließlich von den Komponentenobjekten selbst angestoßen, d.h. ein Script-Objekt für eine Primär- oder Sekundäroperation hat keinen direkten Zugriff auf ein korrespondierendes Komponentenobjekt. Damit wird sichergestellt, daß ein Komponentenobjekt den einzigen Zugangspunkt zu seinem korrespondierenden Objekt darstellt. Diese Eigenschaft erleichtert den Austausch eines korrespondierenden Objekts zur Laufzeit erheblich, da nur eine Referenz auf das korrespondierende Objekt existiert. Der Aufruf einer Sekundäroperation kann zu demselben Interaktionsmuster zwischen den betroffenen Objekten führen, wobei jedoch nicht notwendigerweise eine Aktivierung des korrespondierenden Chassis-

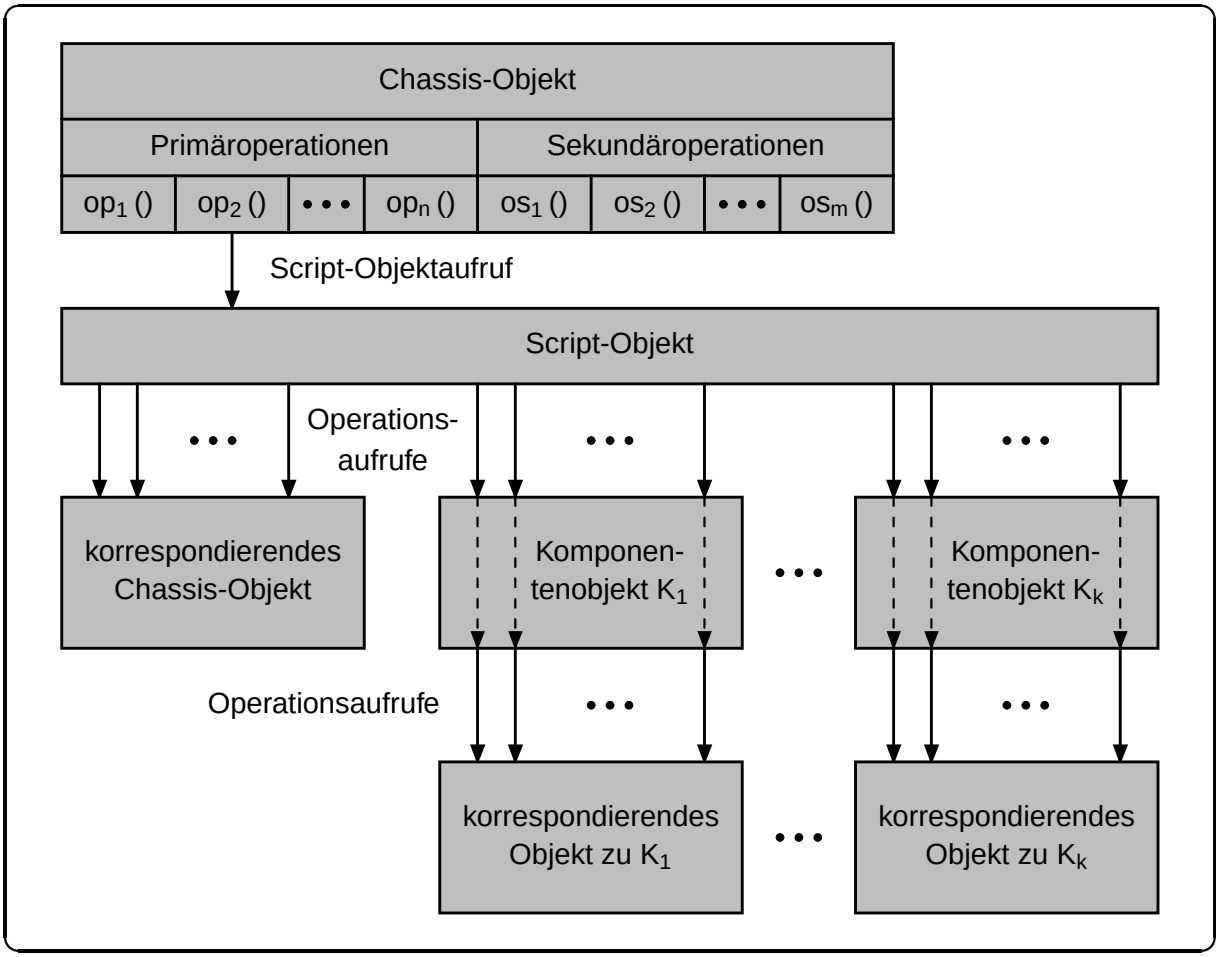


Abb. 162 Script-Objektaufrufe mit Komponentenobjektaufrufen

Objekts durch das Script-Objekt enthalten sein muß.

Der allgemeinste Fall der Ausführung einer Primär- oder Sekundäroperation ist in Abb. 163 (S. 288) dargestellt. Das Script-Objekt ruft hierbei wieder eine oder mehrere Primär- bzw. Sekundäroperationen des Chassis-Objekts auf. Dies hat dann die Aktivierung weiterer Script-Objekte zur Folge.

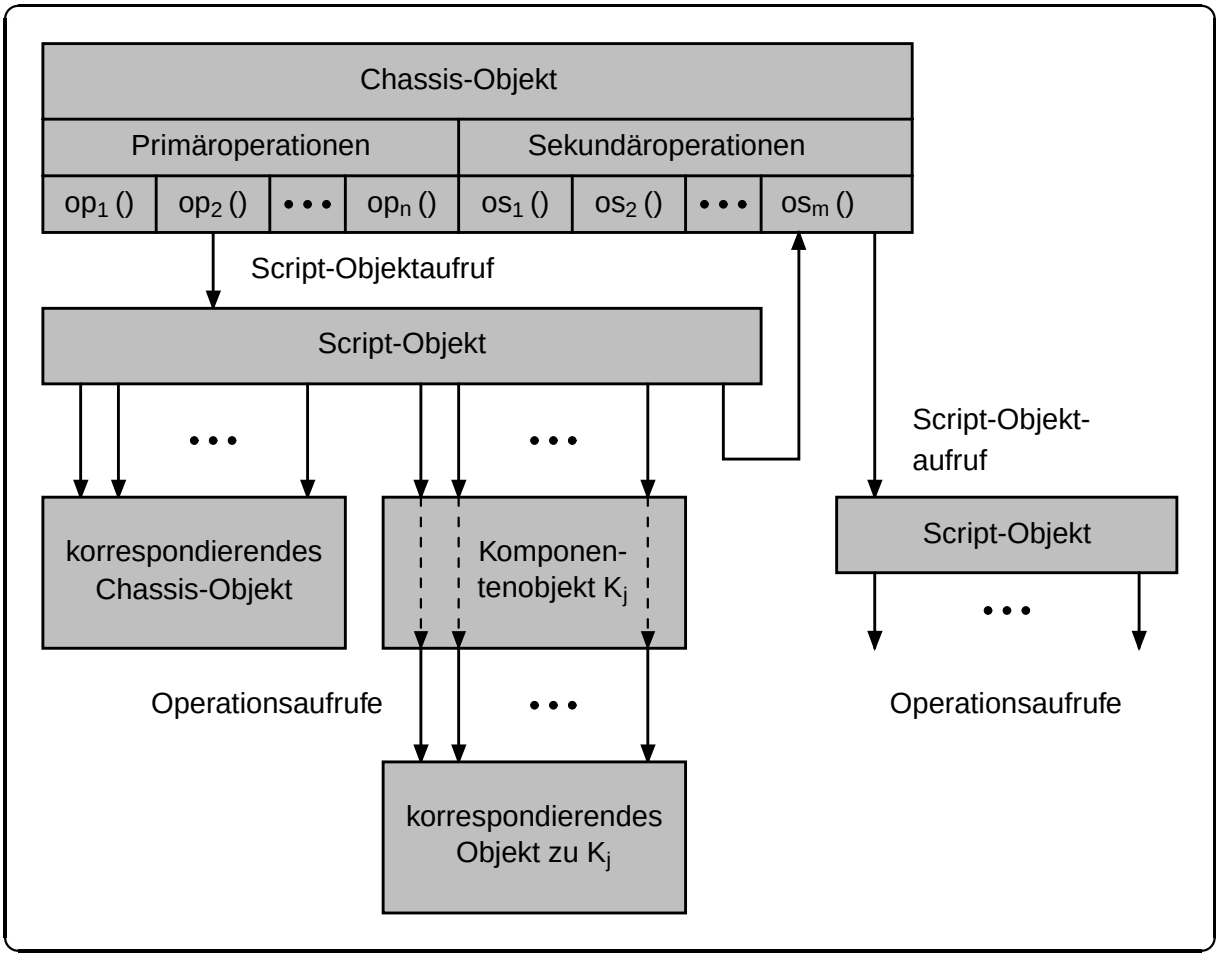


Abb. 163 Script-Objektaufrufe im allgemeinsten Fall

8.3.5 Das dynamische Modell für mehr als zwei Hierarchieebenen

8.3.5.1 Globale Zustandsautomaten

Die globalen Zustandsautomaten der Komponenten- und Chassis-Objekte sind von der Verwendung der Chassis-Komponentenklassen nicht betroffen. Der globale Zustandsautomat einer Chassis-Komponentenklasse stimmt im wesentlichen mit dem Zustandsautomaten aus Abb. 143 (S. 263) überein. Wie bei den Zustandsautomaten aus Abb. 149 (S. 273) und Abb. 150 (S. 273) ist in allen Zuständen die `isDeleted`-Operation hinzuzunehmen, um zu testen, ob die Chassis-Komponente logisch gelöscht ist. Der globale Zustandsautomat einer Chassis-Komponentenklasse ist in Abb. 164 dargestellt.

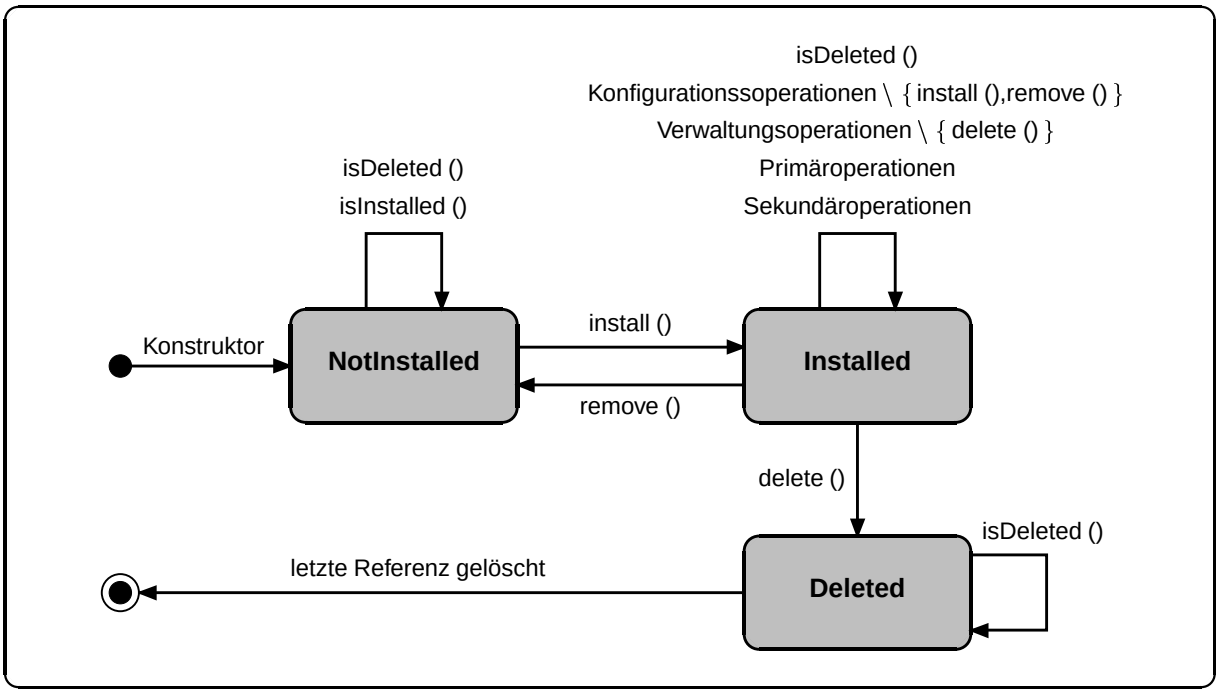


Abb. 164 Der globale Zustandsautomat einer Chassis-Komponentenklasse

Existiert zusätzlich ein lokaler Zustandsautomat, so verfeinert dieser den *Installed*-Zustand.

8.3.5.2 Die Umsetzung der Sequenzdiagramme

Da für Chassis-Komponentenobjekte eigenständige Sequenzdiagramme vorliegen, sind an den Script-Klassen der Chassis-Objekte keine strukturellen Änderungen notwendig.

Die Umsetzung der Sequenzdiagramme beginnt mit der Definition von Script-Interface-Klassen für alle Primäroperationen der Chassis-Komponentenklasse. Ein Interface-Name besitzt denselben Aufbau wie bei den Interface-Klassen für Chassis-Klassen[5] (vgl. S. 275):

- Der erste Namensbestandteil ist der Name der Chassis-Komponentenklasse.
- Es folgt der Operationsname, dessen erster Buchstabe groß geschrieben wird.

[5] Statt eines Chassis-Klassennamens wird natürlich jetzt ein Chassis-Komponentenklassenname verwendet.

- An den Operationsnamen wird noch **Script** angehängt.

Nimmt man die Chassis-Komponentenklasse aus Abb. 146 (S. 267) als Grundlage, so sind fünf Interfaces zu erzeugen:

- **RadioEinschaltenScript**
- **RadioAusschaltenScript**
- **RadioAendernLautstaerkeScript**
- **RadioWechselnFrequenzScript**
- **RadioEntsperrenScript**

Auch die Namensgebung der Script-Klassen für Chassis-Komponentenklassen ist analog zu derjenigen für Chassis-Klassen aufgebaut (vgl. S. 274):

- Der Name beginnt mit dem Namen der Chassis-Komponentenklasse.
- Danach folgt der Name der Komponenten- oder Chassis-Komponentenklasse, zu der die Operation gehört, sofern eine Sekundäroperation vorliegt.
- Der nächste Bestandteil ist der Operationsname, dessen erster Buchstabe groß geschrieben wird.
- Den Abschluß bilden die weiteren Komponenten- und Chassis-Komponentenklassen, die im zugehörigen Sequenzdiagramm aufgeführt sind.

Auf der Basis der Kontrollflußmatrix aus Abb. 145 (S. 265) ergeben sich acht Script-Klassen:

- **RadioEinschalten**
- **RadioEinschaltenCdSpieler**
- **RadioAusschalten**
- **RadioAusschaltenCdSpieler**
- **RadioAendernLautstaerke**
- **RadioWechselnFrequenz**
- **RadioEntsperren**
- **RadioCdSpielerAbspielen**

8.3.5.3 Die Erzeugung eines Chassis-Komponentenobjekts

Die Erzeugung eines Chassis-Komponentenobjekts unterscheidet sich nicht von der Erzeugung eines Chassis-Objekts aus Abschnitt 8.3.4.3 (S. 275):

- Für alle Primäroperationen sind geeignete Script-Objekte auszuwählen.
- Zur Realisierung der Konfigurationsfunktionalität wird eine eigene Konfigurationsklasse verwendet.

In Abb. 165 (S. 291) ist die Struktur der Konfigurationsklasse des **Radio**-Beispiels dargestellt.

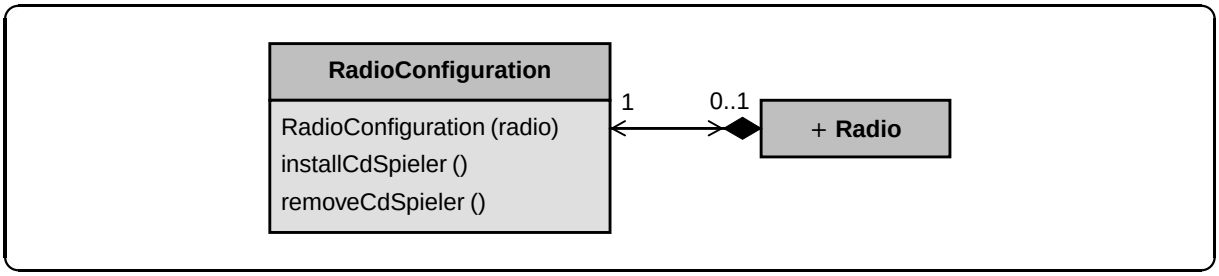


Abb. 165 Die Konfigurationsklasse für ein Radioobjekt

8.3.5.4 Der Aufruf einer Primäroperation

Im Vergleich zum Aufruf einer Primäroperation eines Chassis-Objekts (siehe Abschnitt 8.3.4.4 (S. 279)) muß zusätzlich gelten, daß sich ein Chassis-Komponentenobjekt `cko` im Zustand *Installed* befindet. Damit ergibt sich für den Aufruf einer Primäroperation der folgende Ablauf:

- Test, daß sich das Chassis-Komponentenobjekt nicht bereits im Zustand *Deleted* befindet.
- Überprüfung, ob für das Chassis-Komponentenobjekt der Zustand *Installed* vorliegt.
- Kontrolle, ob das erforderliche Script-Objekt installiert ist.
- Aufruf der Operation des Script-Objekts, die den Kontrollfluß zwischen `cko` und seinen Unterkomponentenobjekten regelt.

Falls einer der drei Tests nicht erfolgreich ist, wird der Aufruf der Primäroperation mit einer entsprechenden Exception abgebrochen. Die Ausführung dieser drei Tests übernimmt die Operation `checkCall`, die der **BaseChassisComponent**-Klasse zugeordnet ist.

8.3.5.5 Der Einbau einer Unterkomponente

Die Bedingungen für den Einbau eines Chassis-Komponentenobjekts in ein Chassis-Objekt entsprechen weitgehend denjenigen für den Einbau eines Komponentenobjekts (siehe S. 280).

- Das Chassis-Objekt muß sich im *Existing*-Zustand befinden.
- Eine entsprechende Chassis-Komponente darf nicht bereits im Chassis-Objekt eingebaut sein.
- Die neue Chassis-Komponente darf nicht gelöscht sein.
- Die neue Chassis-Komponente darf nicht bereits in einem anderen Chassis- oder Chassis-Komponentenobjekt enthalten sein, d.h. sie muß sich im Zustand *NotInstalled* befinden.

Analoge Bedingungen liegen vor, wenn in eine Chassis-Komponente eine Unterkomponente einzubauen ist. Zur Überprüfung der Bedingungen enthält die Klasse **BaseChassisComponent** eine `checkInstall`-Operation. Nach dem Einbau ist möglicherweise eine Anpassung der Kontrollflüsse der Primär- und Sekundäroperationen des Chassis-Komponentenobjekts notwendig. Diese Anpassung wird von dem Konfigurationsobjekt vorgenommen.

Wird beispielsweise in ein **Radio**-Objekt ein CD-Spieler eingebaut, so sind drei neue Script-Objekte erforderlich:

- **RadioEinschaltenCdSpieler** ersetzt **RadioEinschalten**.
- **RadioAusschaltenCdSpieler** ersetzt **RadioAusschalten**.
- **RadioCdSpielerAbspielen** wird neu installiert.

8.3.5.6 Der Aufruf einer Sekundäroperation

Die Aktionen zum Aufruf einer Sekundäroperation eines Chassis-Komponentenobjekts sind identisch mit denen für den Aufruf einer Primäroperation (siehe Abschnitt 8.3.5.4 (S. 291)).

8.3.5.7 Der Ausbau einer Unterkomponente

Sowohl beim Ausbau einer Chassis-Komponente aus einem Chassis-Objekt als auch beim Ausbau einer Unterkomponente aus einem Chassis-Komponentenobjekt sind diesselben Schritte wie beim Ausbau einer Komponente aus einem Chassis-Objekt auszuführen (vgl. S. 284).

Wird aus einem **Radio**-Objekt der CD-Spieler wieder ausgebaut, sind bei den Script-Objekten die folgenden Änderungen notwendig:

- **RadioEinschalten** ersetzt **RadioEinschaltenCdSpieler**.
- **RadioAusschalten** ersetzt **RadioAusschaltenCdSpieler**.
- **RadioCdSpielerAbspielen** wird gelöscht.

8.3.5.8 Das Löschen von Chassis-Komponentenobjekten

Die `delete`-Operation für ein Chassis-Komponentenobjekt hat die folgenden Aktionen auszuführen:

- Das Löschen der Beziehung zu dem korrespondierenden Objekt.
- Aufruf der `delete`-Operation für alle Unterkomponentenobjekte.
- Abbau aller Beziehungen zu den Unterkomponentenobjekten.
- Überführung des Chassis-Komponentenobjekts in den Zustand *Deleted*.

Die `delete`-Operation für ein Chassis-Objekt ist im Vergleich zu der Beschreibung auf S. 285 geringfügig abzuändern: neben den Komponentenobjekten müssen jetzt auch alle Chassis-Komponentenobjekte gelöscht werden.

8.3.5.9 Die Struktur der Klassen BaseChassis und BaseChassisComponent

Die abstrakte Klasse **BaseChassisComponent** stellt eine Kombination der abstrakten Klassen **BaseComponent** und **BaseChassis** dar, wie der Abb. 166 (S. 293) zu entnehmen ist. Da es jedoch für die Erzeugung aussagekräftiger Fehlermeldungen sinnvoll ist, die drei Bauteiltypen **BaseChassis**, **BaseChassisComponent** und **BaseComponent** eindeutig unterscheiden zu können, wird auf die Modellierung einer Vererbungsbeziehung zwischen diesen Klassen verzichtet. Bei einer Implementierung können dann die gemeinsamen Methoden von **BaseChassis** und **BaseChassisComponent** in eine Oberklasse verlagert werden.

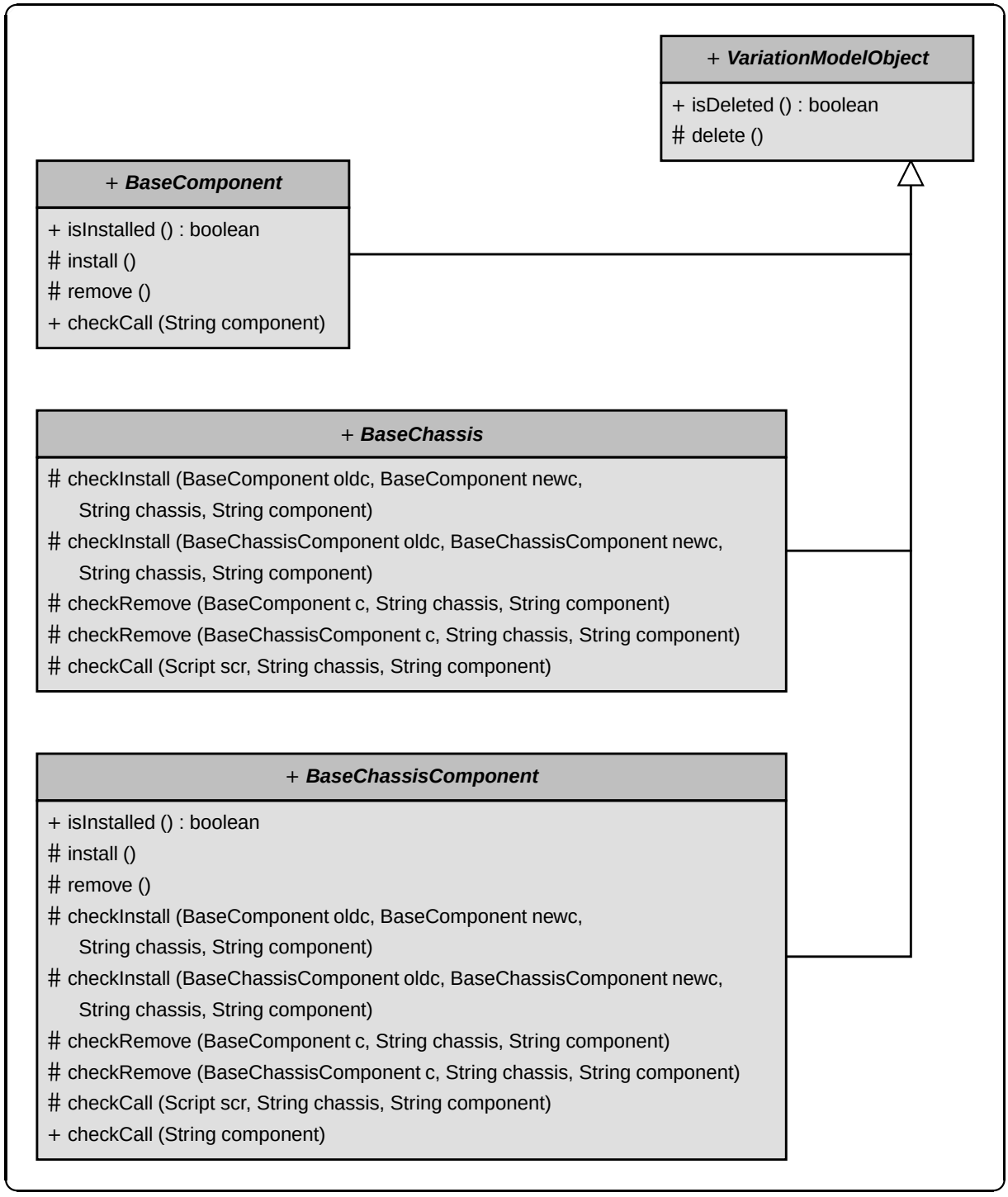


Abb. 166 Die Basisklassen der Bauteiltypen

8.3.5.10 Die Struktur der Script-Klassen

Ein Script-Objekt kann jetzt auch eine Primär- oder Sekundäroperation eines Chassis-Komponentenobjekts aufrufen. Dies führt zur Aktivierung eines weiteren Script-Objekts, wodurch insgesamt eine mehrstufige Struktur der Aufrufbeziehungen entsteht. Abb. 166 zeigt einen Script-Objektaufruf, der sich über (mindestens) drei Ebenen erstreckt.

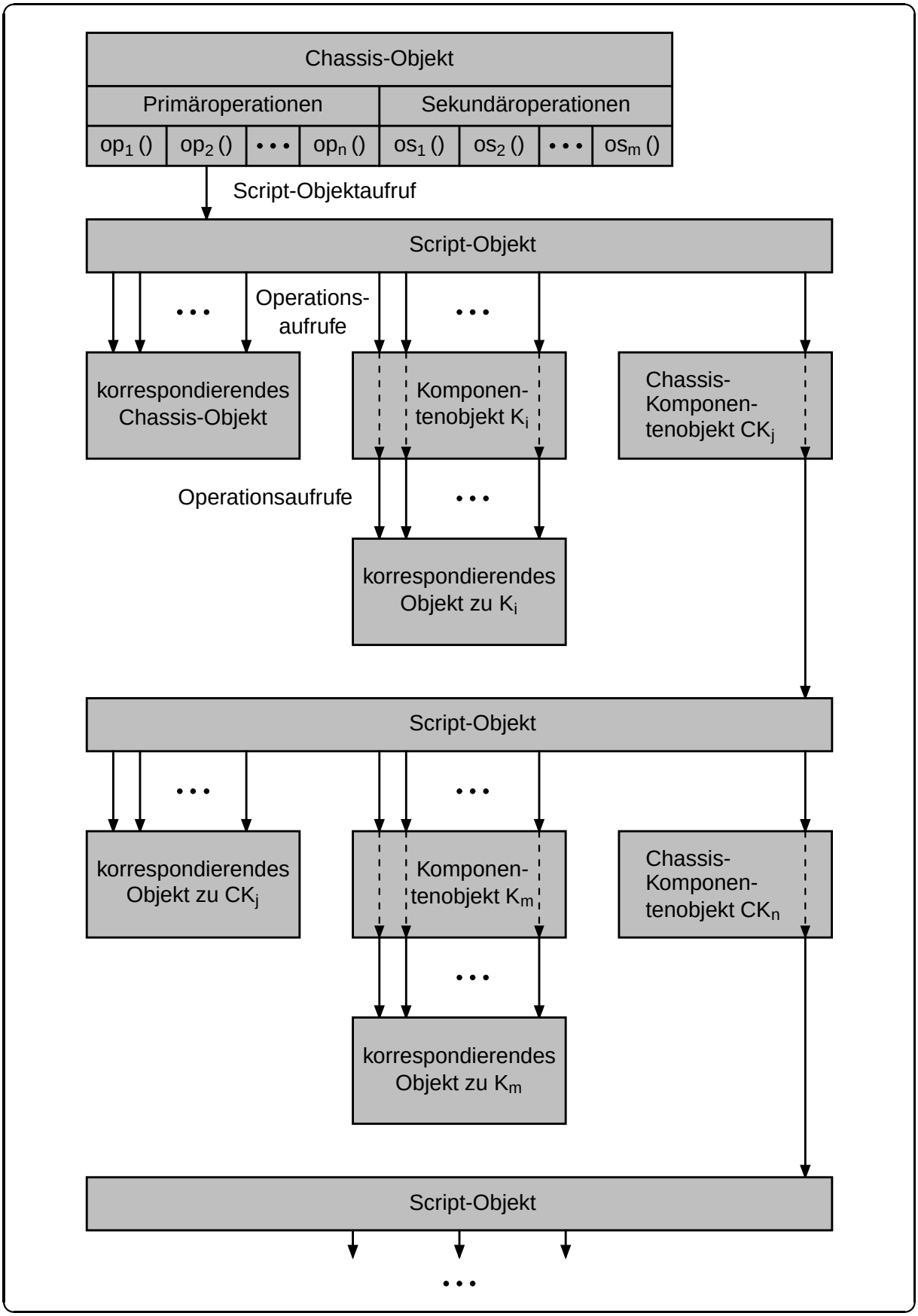


Abb. 167 Script-Objektaufrufe beim Vorhandensein von Chassis-Komponentenobjekten

8.4 Die Generierung von Variantenmodellen

8.4.1 Der Generierungsprozeß

Die Generierung eines Variantenmodells besteht aus fünf Schritten:

- Die Definition der Komponentenhierarchie des Variantenmodells.
- Die Implementierung der zu den Chassis-, ChassisComponent- und Component-Klassen korrespondierenden Klassen, welche die Funktionalität beschreiben.
- Die Generierung selbst.
- Die Implementierung weiterer Script-Klassen, die den Kontrollfluß zwischen den Komponenten festlegen. Innerhalb des Generierungsprozesses werden bereits Script-Klassen erzeugt, die einen Default-Kontrollfluß definieren. Dieser Default-Kontrollfluß besteht lediglich aus dem Aufruf der Operation der korrespondierenden Klasse.
- Die Erstellung der Konfigurationsklassen, die das Konfigurationswissen des Variantenmodells enthalten. Durch den Generierungsprozeß werden Default-Klassen für die Konfiguration erstellt, die als Grundlage für die eigentlichen Konfigurationsklassen verwendbar sind. Die Default-Konfigurationsklassen benutzen die Default-Script-Klassen.

Für die Definition des Variantenmodells ist eine Datei zu erzeugen, deren Inhalt den folgenden Syntaxregeln genügen muß:

```
VariationModelDescription ::= variationmodel$ ComponentDescription*
                              ChassisDescription+ $variationmodel
ChassisDescription        ::= chassis$ ClassIdentifier
                              ComponentPart ExclusionPart^opt
                              $chassis
ComponentDescription      ::= chassiscomponent$ ClassIdentifier
                              ComponentPart ExclusionPart^opt
                              $chassiscomponent
ComponentPart             ::= hascomponents$ Component+
Component                 ::= ClassIdentifier Cardinality^opt
Cardinality               ::= ( Value )
Value                     ::= { 2,3,... }
ExclusionPart             ::= excludes$
                              ⟨ClassIdentifier.MethodIdentifier⟩^opt
ClassIdentifier           ::= UpperCaseLetter
                              ⟨UpperCaseLetter | LowerCaseLetter⟩*
MethodIdentifier          ::= LowerCaseLetter
                              ⟨UpperCaseLetter | LowerCaseLetter⟩*
UpperCaseLetter           ::= { A,B,...,Z }
LowerCaseLetter           ::= { a,b,...,z }
```

Zunächst werden die optional vorhandenen Chassis-Komponenten beschrieben. Damit wird spezifiziert, welche Komponenten (bzw. Chassis-Komponenten) in eine Chassis-Komponente eingebaut werden können (z.B. `CdSpieler` in `Radio`, vgl. Prg. 37 (S. 296)). Für jede einbau-

```
 1: variationmodel$
 2:   chassiscomponent$ Radio
 3:     hascomponents$
 4:       CdSpieler
 5:     excludes$
 6:       CdSpieler.einschalten CdSpieler.ausschalten
 7:   $chassiscomponent
 8:
 9:   chassis$ Auto
10:     hascomponents$
11:       Tuer(4) Radio Schiebedach
12:     excludes$
13:       Radio.entsperren
14:   $chassis
15:
16:   chassis$ RadioTester
17:     hascomponents$
18:       Radio
19:   $chassis
20: $variationmodel
```

Prg. 37 Die Beschreibung des Variantenmodells aus Abschnitt 8.3.3 (S. 270)

bare Komponente kann optional eine Kardinalität ≥ 2 festgelegt werden. Über die **excludes$**-Klausel können Operationen der Unterkomponenten ausgeschlossen werden. Damit läßt sich die Anforderung AV4 (vgl. S. 222) erfüllen. Für das Beispiel aus Prg. 37 gilt daher, daß die Operationen `einschalten` und `ausschalten` des CD-Spielers an der Radioschnittstelle nicht sichtbar sind. Es folgt die Beschreibung der Chassis-Strukturen. Neben der Definition der einbaubaren Komponenten (bzw. Chassis-Komponenten) ist es auch hier wieder mit der **excludes$**-Klausel möglich, Unterkomponentenoperationen auszuschließen. In dem Beispiel aus Prg. 37 wird festgelegt, daß an der Autoschnittstelle die Radiooperation `entsperren` nicht nutzbar ist. Zusätzlich sind die CD-Spieleroperationen `einschalten` und `ausschalten` an der Autoschnittstelle nicht sichtbar, da sie bereits durch die Radiospezifikation ausgeschlossen wurden. Dagegen sind in einem Radiotester alle Operationen des Radios und die nicht ausgeschlossenen Operationen des CD-Spielers verfügbar.

Auf der Grundlage der Variantenmodellbeschreibung aus Prg. 37 und der Klassen **AutoDsc**, **RadioTesterDsc**, **TuerDsc**, **SchiebedachDsc**, **RadioDsc** und **CdSpielerDsc** erzeugt der Generierungsprozeß die folgenden JAVA-Klassen:

- Chassis-Klassen:
  - **Auto**
  - **RadioTester**
- Chassis-Komponentenklasse:
  - **Radio**

- Komponentenklassen:
  - **Schiebedach**
  - **Tuer**
  - **CdSpieler**
- Script-Schnittstellen:
  - **AutoBeschleunigenScript**
  - **AutoBremsenScript**
  - **AutoDruckeFarbeScript**
  - **RadioTesterAusschaltenScript**
  - **RadioTesterEinschaltenScript**
  - **CdSpielerAbspielenScript**
  - **CdSpielerAusschaltenScript**
  - **CdSpielerEinschaltenScript**
  - **RadioAendernLautstaerkeScript**
  - **RadioAusschaltenScript**
  - **RadioEinschaltenScript**
  - **RadioEntsperrenScript**
  - **RadioWechselnFrequenzScript**
  - **SchiebedachOeffnenScript**
  - **SchiebedachSchliessenScript**
  - **TuerAbschliessenScript**
  - **TuerAufschliessenScript**
  - **TuerOeffnenScript**
  - **TuerSchliessenScript**
- Script-Klassen:
  - **AutoBeschleunigen**
  - **AutoBremsen**
  - **AutoDruckeFarbe**
  - **AutoCdSpielerAbspielen**
  - **AutoRadioAendernLautstaerke**
  - **AutoRadioAusschalten**
  - **AutoRadioEinschalten**
  - **AutoRadioWechselnFrequenz**
  - **AutoSchiebedachOeffnen**
  - **AutoSchiebedachSchliessen**
  - **AutoTuerAbschliessen**
  - **AutoTuerAufschliessen**
  - **AutoTuerOeffnen**

  - **AutoTuerSchliessen**
  - **RadioTesterEinschalten**
  - **RadioTesterAusschalten**
  - **RadioTesterRadioAendernLautstaerke**
  - **RadioTesterRadioAusschalten**
  - **RadioTesterRadioEinschalten**
  - **RadioTesterRadioEntsperren**
  - **RadioTesterRadioWechselnFrequenz**
  - **RadioTesterCdSpielerAbspielen**
  - **RadioAendernLautstaerke**
  - **RadioAusschalten**
  - **RadioEinschalten**
  - **RadioEntsperren**
  - **RadioWechselnFrequenz**
  - **RadioCdSpielerAbspielen**
- Konfigurationsklassen:
  - **AutoDefaultConfiguration**
  - **AutoConfiguration**
  - **RadioTesterDefaultConfiguration**
  - **RadioTesterConfiguration**
  - **RadioDefaultConfiguration**
  - **RadioConfiguration**

Die Erzeugung der Script-Schnittstellen findet für alle Primäroperationen aller Klassen statt. Dagegen werden die Script-Klassen nur für die Chassis- und Chassis-Komponentenklassen benötigt. Außerdem werden diejenigen Script-Klassen weggelassen, deren zugeordnete Operationen durch **`excludes$`**-Klauseln ausgeschlossen wurden. Die Konfigurationsklassen sind für Chassis- und Chassis-Komponentenklassen notwendig. Eine Default-Konfigurationsklasse stellt eine Standardkonfiguration zur Verfügung, die sich aus der vorgegebenen Variantenmodellbeschreibung ableiten läßt. Zusätzlich wird zu jeder Default-Konfigurationsklasse eine zweite Konfigurationsklasse generiert. Sie wird nur benötigt, damit das Variantenmodell ohne manuelle Anpassungen übersetzbar ist. Da beispielsweise die Chassis-Klasse **Auto** eine Konfigurationsklasse des Namens **AutoConfiguration** erwartet, wird neben **AutoDefaultConfiguration** die Klasse **AutoConfiguration** erzeugt, die lediglich alle Eigenschaften von **AutoDefaultConfiguration** erbt.

Für eine Anpassung der Standardkonfiguration an die in der Analysephase ermittelten Kontrollflüsse sind schließlich die folgenden manuellen Änderungen notwendig:

- Es müssen entsprechende Script-Klassen erstellt werden.
- Eine neue Konfigurationsklasse ist zu programmieren, die die Änderungen im Vergleich zu der Standardkonfiguration enthält.

In Prg. 38 ist die Struktur der korrespondierenden Klasse **AutoDsc** beschrieben. Die Pro-

```
 1: class AutoDsc {
 2:   private double geschwindigkeit=0.0;
 3:   private String farbe;
 4:   private String typ;
 5:
 6:   public AutoDsc ( String farbe, String typ ) {
 7:     this.farbe = farbe;
 8:     this.typ = typ;
 9:   }
10:
11:   public void druckeFarbe () {
12:     System.out.println ( "Fahrzeug: Typ=" + typ
13:                          + " Farbe=" + farbe );
14:   }
15:
16:   public void bremsen ( double deltaGeschwindigkeit ) {
17:     geschwindigkeit -= deltaGeschwindigkeit;
18:     System.out.println ( "---> bremsen: v=" + geschwindigkeit );
19:   }
20:   public void beschleunigen ( double deltaGeschwindigkeit ) {
21:     geschwindigkeit += deltaGeschwindigkeit;
22:     System.out.println ( "---> beschleunigen: v="
23:                          + geschwindigkeit );
24:   }
25:
26: }
```

Prg. 38 Die korrespondierende Klasse **AutoDsc**

gramme Prg. 39 (S. 300) bis Prg. 41 (S. 302) enthalten einen Auszug der generierten Klasse **Auto**. Das erzeugte Script-Interface **TuerAbschliessenScript** sowie die zugehörige Script-Klasse **AutoTuerAbschliessen** sind in Prg. 42 (S. 303) und Prg. 43 (S. 303) dargestellt. Prg. 44 (S. 303) enthält die manuell erstellte Script-Klasse **AutoTuerAbschliessenSchiebedach**. Hier wird beim Abschließen der Tür auch automatisch das Schiebedach geschlossen. Die zugehörige, generierte Konfigurationsklasse ist in Prg. 45 (S. 304) und Prg. 46 (S. 305) zu sehen, während die manuell erstellte Konfigurationsklasse in Prg. 47 (S. 306) und Prg. 48 (S. 307) dargestellt ist. In der Klasse **AutoDefaultConfiguration** ist beispielsweise die Methode `installTuer` enthalten (vgl. Prg. 45 (S. 304), Zeile 27 bis 37), die die vier Standard-Script-Objekte für eine Tür installiert. In der Klasse **AutoConfiguration** wird das Installieren einer Tür redefiniert (vgl. Prg. 48 (S. 307), Zeile 36 bis 54). Wenn die Tür 0 (d.h. die Fahrertür) betroffen ist und ein Schiebedach vorhanden ist, wird das Script-Objekt der Klasse **AutoTuerAbschliessenSchiebedach** verwendet, um bei einem Abschließen der Tür auch das Schiebedach zu schließen. Entsprechend wird beim Ausbau des Schiebedachs (vgl. Prg. 47 (S. 306), Zeile 24 bis 34) für die Fahrertür wieder das Standard-Script aktiviert.

```
 1: import variationModel.*;
 2:
 3: public final class Auto extends BaseChassis {
 4:
 5:   AutoDsc autoDsc = null;
 6:   private AutoConfiguration autoConfiguration;
 7:   Schiebedach schiebedach = null;
 8:   Tuer[] tuer = new Tuer[4];
 9:   Radio radio =null;
10:
11:   AutoDruckeFarbeScript autoDruckeFarbeScript;
12:   AutoBremsenScript autoBremsenScript;
13:   AutoBeschleunigenScript autoBeschleunigenScript;
14:
15:   TuerOeffnenScript[] tuerOeffnenScript
16:     = new TuerOeffnenScript[4];
17:   TuerSchliessenScript[] tuerSchliessenScript
18:     = new TuerSchliessenScript[4];
19:   TuerAufschliessenScript[] tuerAufschliessenScript
20:     = new TuerAufschliessenScript[4];
21:   TuerAbschliessenScript[] tuerAbschliessenScript
22:     = new TuerAbschliessenScript[4];
23:
24:   SchiebedachOeffnenScript schiebedachOeffnenScript;
25:   SchiebedachSchliessenScript schiebedachSchliessenScript;
26:
27:   RadioEinschaltenScript radioEinschaltenScript;
28:   RadioAusschaltenScript radioAusschaltenScript;
29:   RadioAendernLautstaerkeScript radioAendernLautstaerkeScript;
30:   RadioWechselnFrequenzScript radioWechselnFrequenzScript;
31:   CdSpielerAbspielenScript cdSpielerAbspielenScript;
32:
33:   public Auto (String farbe,String typ) {
34:     autoConfiguration = new AutoConfiguration (this);
35:     autoDsc = new AutoDsc (farbe,typ);
36:   }
37:
38:   public void delete()
39:          throws ChassisDeleted, ComponentDeleted,
40:                 ChassisComponentDeleted {
41:     autoDsc = null;
42:     if ( schiebedach!=null ) {
43:        if ( !schiebedach.isDeleted() )
44:           schiebedach.delete();
45:        schiebedach = null;
46:     }
```

Prg. 39 Die generierte Klasse **Auto** – Teil 1

```
47:      for ( int i=0; i<tuer.length; i++ ) {
48:        if ( tuer[i]!=null )
49:           if ( !tuer[i].isDeleted() )
50:              tuer[i].delete();
51:           tuer[i] = null;
52:      }
53:      if ( radio != null ) {
54:        if ( !radio.isDeleted() ) {
55:           radio.delete();
56:        }
57:        radio = null;
58:      }
59:      super.delete();
60:    }
61:
62:    public void installSchiebedach (Schiebedach schiebedach)
63:           throws VariationModelException {
64:      checkInstall (this.schiebedach,schiebedach,"Auto",
65:                    "Schiebedach");
66:      this.schiebedach = schiebedach;
67:      autoConfiguration.installSchiebedach();
68:      schiebedach.install();
69:    }
70:
71:    public void removeSchiebedach()
72:           throws VariationModelException {
73:      checkRemove (schiebedach,"Auto","Schiebedach");
74:      autoConfiguration.removeSchiebedach();
75:      schiebedach.remove();
76:      this.schiebedach = null;
77:    }
78:
79:    public boolean hasSchiebedach ()
80:           throws ChassisDeleted {
81:      if ( isDeleted() )
82:         throw new ChassisDeleted ("Auto");
83:      return schiebedach!=null;
84:    }
85:
86:    public void installTuer (Tuer tuer,int nr)
87:           throws VariationModelException {
88:      checkInstall (this.tuer[nr],tuer,"Auto","Tuer["+nr+"]");
89:      this.tuer[nr]=tuer;
90:      autoConfiguration.installTuer (nr);
91:      tuer.install();
92:    }
```

Prg. 40 Die generierte Klasse **Auto** – Teil 2

```
 93:    ...
 94:
 95:    public void druckeFarbe ()
 96:           throws VariationModelException {
 97:      checkCall (autoDruckeFarbeScript,"Auto",
 98:                 "AutoDruckeFarbeScript");
 99:      autoDruckeFarbeScript.druckeFarbe();
100:    }
101:
102:    public void bremsen (double deltaGeschwindigkeit)
103:           throws VariationModelException {
104:      checkCall (autoBremsenScript, "Auto",
105:                 "AutoBremsenScript");
106:      autoBremsenScript.bremsen (deltaGeschwindigkeit);
107:    }
108:
109:    public void beschleunigen (double deltaGeschwindigkeit)
110:           throws VariationModelException {
111:      checkCall (autoBeschleunigenScript,"Auto",
112:                 "AutoBeschleunigenScript");
113:      autoBeschleunigenScript.beschleunigen (deltaGeschwindigkeit);
114:    }
115:
116:    public void oeffnenSchiebedach()
117:           throws VariationModelException {
118:      checkCall (schiebedachOeffnenScript,"Auto",
119:                  "SchiebedachOeffnenScript");
120:      schiebedachOeffnenScript.oeffnen();
121:    }
122:
123:    public void schliessenSchiebedach()
124:           throws VariationModelException {
125:      checkCall (schiebedachSchliessenScript,"Auto",
126:                 "SchiebedachSchliessenScript");
127:      schiebedachSchliessenScript.schliessen();
128:    }
129:
130:    public void oeffnenTuer (int nr)
131:           throws VariationModelException {
132:      checkCall (tuerOeffnenScript[nr],"Auto",
133:                 "TuerOeffnenScript");
134:      tuerOeffnenScript[nr].oeffnen();
135:    }
136:
137:    ...
138:
139:  }
```

Prg. 41 | Die generierte Klasse **Auto** – Teil 3

```
 1: import variationModel.*;
 2: interface TuerAbschliessenScript extends Script {
 3:   void abschliessen ()
 4:        throws VariationModelException;
 5: }
```

Prg. 42 Die generierte Schnittstelle **TuerAbschliessenScript**

```
 1: import variationModel.*;
 2: class AutoTuerAbschliessen implements TuerAbschliessenScript {
 3:   private Tuer tuer;
 4:   AutoTuerAbschliessen (Tuer tuer) {
 5:     this.tuer = tuer;
 6:   }
 7:   public void abschliessen()
 8:          throws VariationModelException {
 9:     tuer.checkCall ("Tuer");
10:     tuer.abschliessen();
11:   }
12: }
```

Prg. 43 Die generierte Klasse **AutoTuerAbschliessen**

```
 1: import variationModel.*;
 2: class AutoTuerAbschliessenSchiebedach
 3:       implements TuerAbschliessenScript {
 4:   private Tuer tuer;
 5:   private Schiebedach schiebedach;
 6:
 7:   AutoTuerAbschliessenSchiebedach
 8:       (Tuer tuer,Schiebedach schiebedach) {
 9:     this.tuer = tuer;
10:     this.schiebedach = schiebedach;
11:   }
12:
13:   public void abschliessen ()
14:          throws VariationModelException {
15:     tuer.checkCall ("Tuer");
16:     schiebedach.checkCall ("Schiebedach");
17:     tuer.abschliessen();
18:     schiebedach.schliessen();
19:   }
20: }
```

Prg. 44 Die manuell erstellte Klasse **AutoTuerAbschliessenSchiebedach**

```
 1: import variationModel.*;
 2:
 3: class AutoDefaultConfiguration {
 4:
 5:   protected Auto auto;
 6:   AutoDefaultConfiguration (Auto auto) {
 7:     this.auto = auto;
 8:     auto.autoDruckeFarbeScript = new AutoDruckeFarbe (auto);
 9:     auto.autoBremsenScript = new AutoBremsen (auto);
10:     auto.autoBeschleunigenScript = new AutoBeschleunigen (auto);
11:   }
12:
13:   void installSchiebedach()
14:        throws ChassisDeleted, ComponentDeleted {
15:     auto.schiebedachOeffnenScript
16:          = new AutoSchiebedachOeffnen (auto.schiebedach);
17:     auto.schiebedachSchliessenScript
18:          = new AutoSchiebedachSchliessen (auto.schiebedach);
19:   }
20:
21:   void removeSchiebedach()
22:        throws ChassisDeleted, ComponentDeleted {
23:     auto.schiebedachOeffnenScript = null;
24:     auto.schiebedachSchliessenScript = null;
25:   }
26:
27:   void installTuer (int nr)
28:        throws ChassisDeleted, ComponentDeleted {
29:     auto.tuerAbschliessenScript[nr]
30:          = new AutoTuerAbschliessen (auto.tuer[nr]);
31:     auto.tuerOeffnenScript[nr]
32:          = new AutoTuerOeffnen (auto.tuer[nr]);
33:     auto.tuerSchliessenScript[nr]
34:          = new AutoTuerSchliessen (auto.tuer[nr]);
35:     auto.tuerAufschliessenScript[nr]
36:          = new AutoTuerAufschliessen (auto.tuer[nr]);
37:   }
38:
39:   void removeTuer (int nr)
40:        throws ChassisDeleted, ComponentDeleted {
41:     auto.tuerOeffnenScript = null;
42:     auto.tuerSchliessenScript = null;
43:     auto.tuerAufschliessenScript = null;
44:     auto.tuerAbschliessenScript = null;
45:   }
```

Prg. 45 Die generierte Klasse **AutoDefaultConfiguration** – Teil 1

```
46:
47:    void installRadio()
48:         throws ChassisDeleted, ChassisComponentDeleted {
49:      auto.radioEinschaltenScript
50:           = new AutoRadioEinschalten (auto.radio);
51:      auto.radioAusschaltenScript
52:           = new AutoRadioAusschalten (auto.radio);
53:      auto.radioAendernLautstaerkeScript
54:           = new AutoRadioAendernLautstaerke (auto.radio);
55:      auto.radioWechselnFrequenzScript
56:           = new AutoRadioWechselnFrequenz (auto.radio);
57:      auto.cdSpielerAbspielenScript
58:           = new AutoRadioAbspielen (auto.radio);
59:    }
60:
61:    void removeRadio()
62:         throws ChassisDeleted, ChassisComponentDeleted {
63:      auto.radioEinschaltenScript = null;
64:      auto.radioAusschaltenScript = null;
65:      auto.radioAendernLautstaerkeScript = null;
66:      auto.radioWechselnFrequenzScript = null;
67:      auto.cdSpielerAbspielenScript = null;
68:    }
69:
70:  }
```

Prg. 46 Die generierte Klasse **AutoDefaultConfiguration** – Teil 2

```
 1: import variationModel.*;
 2:
 3: class AutoConfiguration extends AutoDefaultConfiguration {
 4:
 5:   AutoConfiguration ( Auto auto ) {
 6:       super (auto);
 7:   }
 8:
 9:   void installSchiebedach ()
10:        throws ChassisDeleted, ComponentDeleted {
11:     if ( auto.hasTuer(0) ) {
12:        if ( auto.tuer[0].isDeleted() )
13:           throw new ComponentDeleted ( "Tuer[0] is deleted" );
14:        auto.tuerAbschliessenScript[0]
15:             = new AutoTuerAbschliessenSchiebedach
16:                   (auto.tuer[0], auto.schiebedach );
17:     }
18:     auto.schiebedachOeffnenScript
19:          = new AutoSchiebedachOeffnen (auto.schiebedach);
20:     auto.schiebedachSchliessenScript
21:          = new AutoSchiebedachSchliessen (auto.schiebedach);
22:   }
23:
24:   void removeSchiebedach ()
25:        throws ChassisDeleted, ComponentDeleted {
26:     if ( auto.hasTuer(0) ) {
27:        if ( auto.tuer[0].isDeleted() )
28:           throw new ComponentDeleted ( "Tuer[0] is deleted" );
29:        auto.tuerAbschliessenScript[0]
30:             = new AutoTuerAbschliessen (auto.tuer[0]);
31:     }
32:     auto.schiebedachOeffnenScript = null;
33:     auto.schiebedachSchliessenScript = null;
34:   }
```

Prg. 47 Die manuell erstellte Klasse **AutoConfiguration** – Teil 1

```
35:
36:    void installTuer ( int nr )
37:      throws ChassisDeleted, ComponentDeleted {
38:      if ( nr==0 && auto.hasSchiebedach() ) {
39:
40:         auto.tuerAbschliessenScript[nr] =
41:              new AutoTuerAbschliessenSchiebedach
42:                  (auto.tuer[nr], auto.schiebedach );
43:      }
44:      else {
45:         auto.tuerAbschliessenScript[nr] =
46:              new AutoTuerAbschliessen (auto.tuer[nr]);
47:      }
48:      auto.tuerOeffnenScript[nr]
49:           = new AutoTuerOeffnen (auto.tuer[nr]);
50:      auto.tuerSchliessenScript[nr]
51:           = new AutoTuerSchliessen (auto.tuer[nr]);
52:      auto.tuerAufschliessenScript[nr] =
53:           new AutoTuerAufschliessen (auto.tuer[nr]);
54:    }
55:
56:    void removeTuer ( int nr ) {
57:      auto.tuerOeffnenScript = null;
58:      auto.tuerSchliessenScript = null;
59:      auto.tuerAufschliessenScript = null;
60:      auto.tuerAbschliessenScript = null;
61:    }
62:
63:  }
```

Prg. 48 Die manuell erstellte Klasse **AutoConfiguration** – Teil 2

Die Struktur des Pakets **variationModel** ist in Abb. 168 (S. 308) dargestellt. Alle Script-Klassen erweitern das Interface **Script**. Benötigt eine korrespondierende Klasse eigene Exception-Klassen, so müssen diese von **OperationalException** abgeleitet werden. Die restlichen Exception-Klassen werden in den Konfigurationsklassen und den Unterklassen von **VariationModelObject** verwendet.

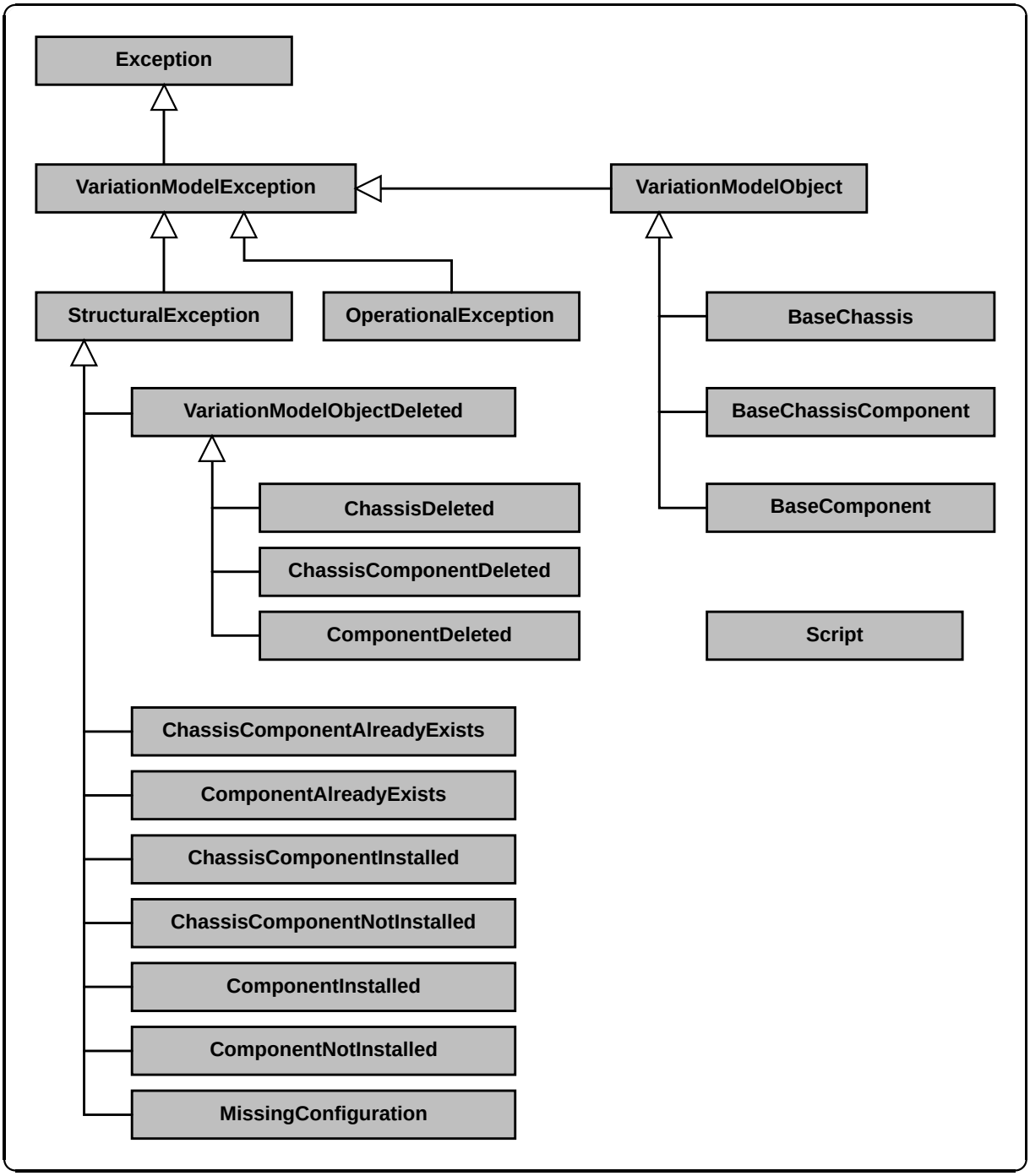


Abb. 168 Das Paket **variationModel**

8.4.2 Die Software-Architektur zur Generierung von Variantenmodellen

Das statische Entwurfsmodell des Variantenmodellgenerators aus Abb. 169 (S. 310) baut auf der Struktur des entsprechenden Entwurfsmodells des Rollenmodellgenerators (vgl. Abb. 101 (S. 198)) auf. Um den Generierungsprozeß zu starten, wird die `main`-Methode der Klasse **VariationModelUI** aufgerufen. Diese bekommt als Parameter den Namen der Datei übergeben, die die Beschreibung des Variantenmodells enthält. Die Generierung besteht aus den folgenden Schritten:[6]

- Das **VariationModelParser**-Objekt führt auf der Basis des vom **Scanner**-Objekt erzeugten **Token**-Arrays[7] die Syntaxprüfung durch. Falls die Syntaxprüfung erfolgreich war, liefert sie als Ergebnis ein **VariationModelDescription**-Objekt zurück.
- Ein **VariationModelDescription**-Objekt enthält alle für den Generierungsprozeß erforderlichen Informationen. Die Chassis-, ChassisComponent- und Component-Beschreibungen werden durch **ComponentDescription**-Objekte repräsentiert.
- Da eine Variantenmodellstruktur wieder einen azyklischen Graphen darstellt, stimmt die **ComponentHierarchy**-Klasse in wesentlichen Teilen mit der **InheritanceStructure**-Klasse des Rollenmodellgenerators überein. Falls der Graph nicht zyklenfrei ist, wird der Generierungsprozeß abgebrochen.
- Nach der Erzeugung eines **VariationModelGenerator**-Objekts werden über den Aufruf der `generate`-Methode alle Klassen des Variantenmodells erzeugt.

[6] Auf eine weitergehende Beschreibung der einzelnen Klassen des Entwurfsmodells wird hier verzichtet, da in vielen Punkten eine sehr enge Übereinstimmung mit den Strukturen aus dem Entwurfmodell des Rollenmodellgenerators besteht.

[7] Das Interface **Keywords** wurde hierfür um die Schlüsselwörter der Variantenmodellbeschreibung ergänzt.

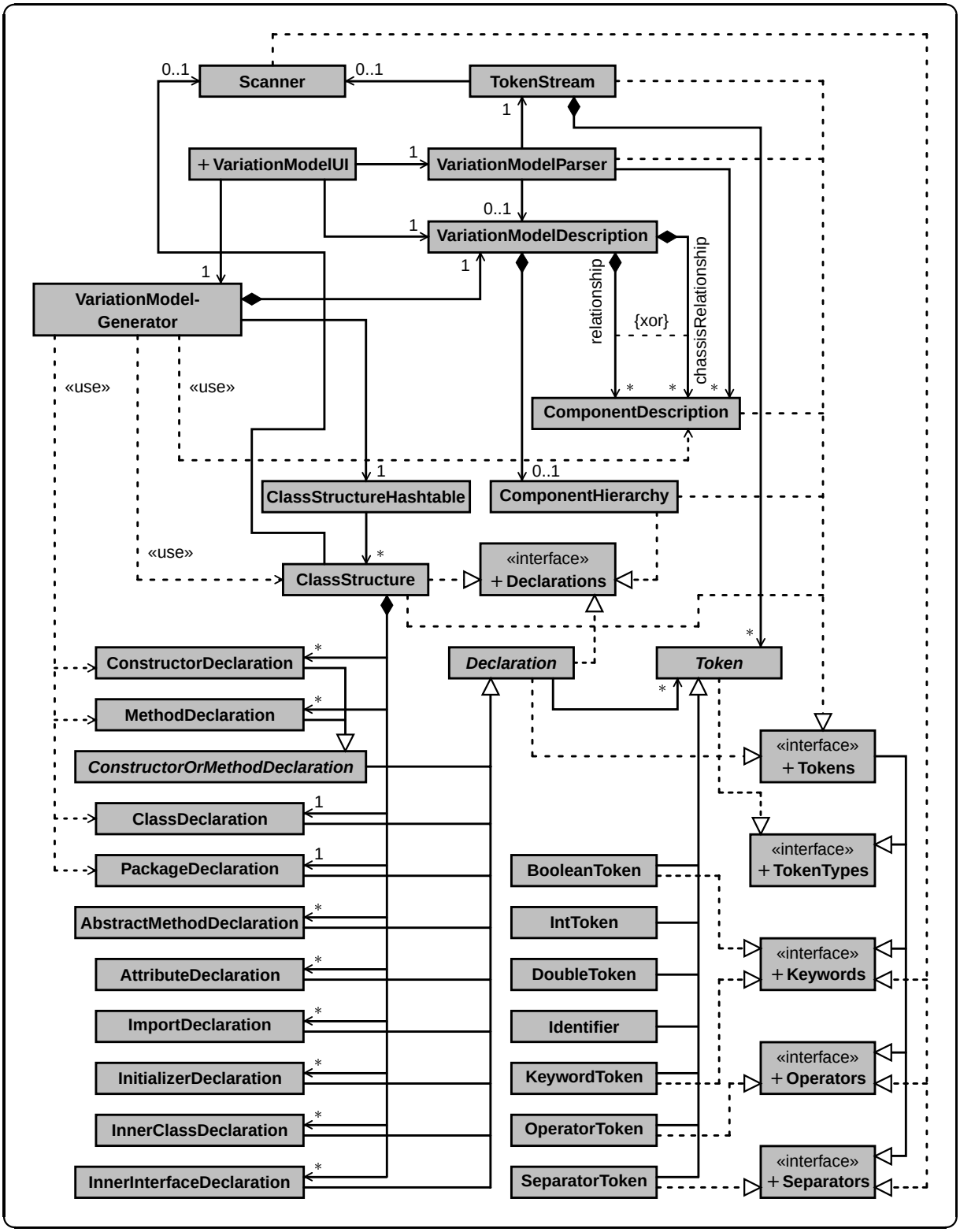


Abb. 169 Das Entwurfsmodell des Variantenmodellgenerators

8.5 Die Integration des Variantenmodells in den Software-Entwicklungsprozeß

8.5.1 Der Analyseprozeß

Die Entwicklung des statischen Analysemodells finden in drei Schritten statt.

- Zunächst sind die Chassis-, Chassis-Komponenten- und Komponentenklassen zusammen mit ihren Attributen und ihren nach außen sichtbaren Operationen zu identifizieren. Damit liegt die Struktur der korrespondierenden Klassen des Variantenmodells fest.
- Im zweiten Schritt ist zu ermitteln, welche Komponenten in welche anderen Komponenten eingebaut werden können.
- Im dritten Schritt wird spezifiziert, welche Operationen der untergeordneten Komponenten an den Schnittstellen der Chassis- und Chassis-Komponentenobjekte sichtbar sein sollen. Diese Operationen werden den entsprechenden Klassen hinzugefügt.

Für das dynamische Analysemodell stellt die Spezifikation der Sequenzdiagramme die zentrale Aufgabe dar. Aus ihnen lassen sich die globalen Kontrollflüsse und das Konfigurationswissen ableiten.

8.5.2 Das Entwurfsmodell

Die in der Analysephase ermittelten Klassen stellen den Ausgangspunkt für die Applikationsschicht einer Drei-Schichten-Architektur dar. Die korrespondierenden Klassen aus dem Analysemodell müssen um entsprechende Operationen zur Kommunikation mit der Präsentationsschicht ergänzt werden. Außerdem ist die Schnittstelle zu der Datenzugriffsschicht zu spezifizieren.

8.5.3 Die Implementierungsphase

Zunächst sind die korrespondierenden Klassen aus dem Entwurfsmodell zu implementieren. Nachdem die Beschreibungsdatei für das Variantenmodell erstellt wurde, kann die Grundversion des Variantenmodells generiert werden. Danach müssen die Konfigurations- und Script-Klassen implementiert werden, welche das für ein Variantenmodell spezifische Wissen enthalten. Diese Klassen werden von den generierten Klassen abgeleitet.

8.6 Kapitelzusammenfassung

Das vorgestellte Variantenmodell erfüllt die Anforderungen AV1 bis AV6 (vgl. S. 222) durch die Trennung der Aspekte ***Strukturbeschreibung***, ***Objektfunktionalität***, ***globaler Kontrollfluß*** und ***Konfigurationswissen***.

Für die Beschreibung der Struktur eines Variantenmodells wurde eine Grammatik definiert, mit der sich die Anforderungen AV1, AV2 und AV4 umsetzen lassen. Zum einen wird über eine Strukturbeschreibung festgelegt, welche Komponenten in welche anderen Komponenten eingebaut werden können. Da die Grammatik die Spezifikation von Variantenmodellen mit einer beliebigen Hierarchiestufenanzahl erlaubt, sind die Anforderungen AV1 und AV2 erfüllt. Zum anderen kann über die Strukturbeschreibung spezifiziert werden, welche Operationen an der Schnittstelle einer Chassis- bzw. Chassis-Komponentenklasse sichtbar sind. Als Grundlage wird zunächst angenommen, daß alle identifizierten Operationen der Komponenten auch zur Schnittstelle des übergeordneten Objekts gehören. Durch die **`excludes$`**-Klausel können dann einzelne Operationen ausgeschlossen werden. Damit ist die Anforderung AV4 umgesetzt.

Die Spezifikation der Objektfunktionalität erfolgt über eine korrespondierende Klasse. Hier werden die Operationen definiert und implementiert, die den internen Zustand einer Komponente betreffen. Die korrespondierenden Klassen bilden zusammen mit der Strukturbeschreibung den Ausgangspunkt für die Generierung des Variantenmodells. Zu jeder korrespondierenden Klasse wird eine Zugriffsklasse erzeugt. Die Chassis- und Chassis-Komponentenzugriffsklassen besitzen entsprechende install- und remove-Operationen für den Ein- und Ausbau der Unterkomponentenobjekte. Um die Anforderung AV3 zu erfüllen, daß eine Komponente physikalisch maximal Bestandteil eines Chassis sein kann, wurden Zustandsautomaten für Chassis-, Chassis-Komponenten und Komponentenklassen definiert, deren Funktionalität bei der Generierung in die entsprechenden Klassen integriert wird. Diese Zustandsautomaten überwachen den Ein- und Ausbau von Komponenten.

Die Anforderungen AV5 und AV6 betreffen die globalen Kontrollflüsse zwischen den beteiligten Objekten und deren Konfigurierbarkeit. Um die globalen Kontrollflüsse in einem konventionellen Ansatz von der aktuellen Konfiguration abhängig zu machen, wird man typischerweise `if`- und `switch`-Anweisungen innerhalb der vom globalen Kontrollfluß betroffenen Operationen verwenden. Dies entspricht einer Verteilung des Konfigurationswissens über die einzelnen Objekte. Damit werden die globalen Kontrollflüsse schwer nachvollziehbar und es ist sehr aufwendig, Änderungen vorzunehmen, wenn beispielsweise in einer neuen Version der Chassis-Klasse neue Unterkomponenten integriert werden sollen. Aus diesem Grund wurde in dem entwickelten Modell das Konfigurationswissen in einer eigenen Konfigurationsklasse konzentriert. Jedes Chassis-Objekt besitzt dann ein eigenes Konfigurationsobjekt. Dieses stellt die Funktionalität der install- und remove-Operationen zur Verfügung, und legt fest, welche Kontrollflüsse durch eine Konfigurationsänderung betroffen sind. Ebenso wie das Konfigurationswissen wurden die globalen Kontrollflüsse in eigenen Objekten gekapselt, den Script-Objekten. Diese können als Implementierungen von Sequenzdiagrammen angesehen werden, da sie zum einen die Referenzen auf die betroffenen Objekte enthalten und zum anderen die Operationsaufrufe.

Weitere wichtige Eigenschaften des vorgestellten Ansatzes sind:

- Die spezifizierte Beschreibungssprache läßt sich als Aspektbeschreibungssprache auffassen, die den *Aspekt Variantenmodellstruktur* betrifft.

- Die korrespondierenden Klassen können in unterschiedlichen Variantenmodellen wiederverwendet werden, da sie selbst keine expliziten Beziehungen zu (anderen) korrespondierenden Klassen besitzen. Auf diesem Weg ist es sehr einfach, unterschiedliche Ausprägungen eines Variantenmodells zu erzeugen.
- Durch die Generierung spezieller Verwaltungsoperationen für die beteiligten Klassen und die Verwendung der Sprache JAVA wird eine vollständige Typsicherheit erreicht.
- Die Verwaltungsoperationen erzeugen im Fehlerfall entsprechende Exception-Objekte. Damit wird der Anwendungsentwickler gezwungen, auf Fehler zu reagieren, was die Software-Qualität erhöht.

Nach der Vorstellung des Analyse- und des Entwurfsmodells für die Umsetzung des neuen Variantenmodells wurde die Software-Architektur zur Generierung des Variantenmodells beschrieben, die wesentliche Teile der Software-Architektur des Rollenmodellgenerators wiederverwendet. Den Abschluß des Kapitels stellte die Integration des Variantenmodells in den Software-Entwicklungsprozeß dar. Hier wurden aus der Sicht des Anwendungsentwicklers die notwendigen Schritte zur Realisierung eines Variantenmodells zusammengefaßt.

Kapitel 9

Fallstudie: Ein Variantenmodell für die Scheduling-Komponente eines dynamisch adaptiven Betriebssystems

9.1 Einleitung

In den letzten Jahren ist innerhalb der Betriebssystemforschung die Forderung nach *adaptionsfähigen* Betriebssystemstrukturen zunehmend in den Vordergrund gerückt. Diese Notwendigkeit steht in engem Zusammenhang mit der inhärenten Eigenschaft eines Betriebssystems, einerseits eine direkte Schnittstelle zur Hardware zu besitzen und andererseits den Applikationen geeignete *Abstraktionen* der verwalteten Hardware-Betriebsmittel anzubieten. Beide Schnittstellen zeichnen sich durch eine vergleichsweise hohe Instabilität aus:

- Die Anforderungsprofile der Applikationen bezüglich der benötigten Betriebssystemfunktionalitäten werden immer *differenzierter*.
- Die Entwicklung auf dem Hardware-Sektor verlangt entsprechende Anpassungen innerhalb des Betriebssystems.

Die Verwendung eines Adaptionsprozesses hat immer das Ziel, die vorhandene Systemstruktur zu optimieren. Innerhalb eines Betriebssystems gibt es hierfür sehr viele Ansatzpunkte. Das Spektrum erstreckt sich von einer reinen Parameteranpassung (z.B. Neuberechnung des Zeitscheibenquantums) über die Auswahl oder Erzeugung optimaler Betriebsmittelverwaltungsstrategien bis hin zur Anpassung der prinzipiellen Systemstruktur. Der letzte Ansatzpunkt hat das Ziel, einer Menge von Applikationen *genau* und *ausschließlich* diejenigen Dienstleistungen bereitzustellen, die von diesen Applikationen benötigt werden (***Erzeugung einer minimalen Betriebssystemstruktur***). Zusätzliche Betriebssystemfunktionalitäten verbrauchen Systemresourcen (z.B. Hauptspeicher und Rechenleistung), ohne für die Applikation einen Nutzen zu haben. Die Konstruktion minimaler Betriebssystemstrukturen spielte eine zentrale Rolle innerhalb des PEACE-Projekts [SP94]. Sind die Anforderungen der auf dem Betriebssystem auszuführenden Applikationen bekannt und konstant, so kann eine für diese Applikationsmenge

maßgeschneiderte Betriebssystemarchitektur entworfen und implementiert werden (***statische Adaption***). Sind die Anforderungsprofile im voraus nicht bekannt, so ist eine Optimierung der Betriebssystemstruktur nur durch die Adaption des Systems zur Laufzeit erreichbar (***dynamische Adaption***). Für die Erzeugung minimaler Betriebssystemstrukturen bietet sich zunächst die statische Adaption an, da hier die Anpassung bereits auf der Entwurfsebene stattfinden kann. Bei der dynamischen Adaption konzentrieren sich existierende Ansätze auf das Laden und Entfernen kompletter Systemteile, wobei die Zielsetzung eher in der Anpassung an eine veränderte Hardware-Struktur liegt (z.B. dynamisches Laden von Treibermodulen) als in der Erzeugung von Systemstrukturen, die aus dem Blickwinkel der Applikationen als minimal angesehen werden können. Das neue Modell zur Erzeugung minimaler applikationsspezifischer Betriebssystemstrukturen durch dynamische Adaption wurde in [BF99] vorgestellt.

In diesem Kapitel werden zunächst unterschiedliche Scheduling-Modelle beschrieben. Für diese Modelle erfolgt eine Analyse der globalen Kontrollflüsse, die die Grundlage für eine dynamische Adaption darstellen. Das abgeleitete Modell einer minimalen Scheduling-Komponente wird dann auf ein Variantenmodell abgebildet.

9.2 Scheduling-Modelle

Für die Betriebssystemkomponente *Scheduling* wurden sehr viele strukturell unterschiedliche Modelle implementiert. Die gemeinsame Grundlage für die hier diskutierten Modelle bilden die Betriebssystemobjekte der Klassen **Process** und **Thread**. Damit können beim Scheduling die Operationen *Thread-Wechsel* und *Prozeßwechsel* getrennt behandelt werden. Ein Thread-Wechsel hat dann nicht notwendigerweise automatisch einen *Adreßraumwechsel* zur Folge. Tab. 4 charakterisiert vier Scheduling-Modelle über die jeweils erlaubten Prozeß- und Thread-Anzahlen. SM1 stellt das einfachste Modell dar. Prozeß- und Thread-Wechsel sind nicht vor-

Modell	Prozeßanzahl im Betriebssystem	Thread-Anzahl innerhalb eines Prozesses
SM1	1	1
SM2	1	n ($>$1)
SM3	m ($>$1)	1
SM4	m ($>$1)	n ($>$1)

Tab. 4 Scheduling-Modelle

gesehen, so daß die Betriebssystemobjekte *ProcessScheduler* und *ThreadScheduler* nicht benötigt werden. MS-DOS kann als Vertreter dieses Modells angesehen werden, auch wenn dort die Begriffe *Prozeß* und *Thread* nicht verwendet werden. Das Modell SM2 läßt sich beispielsweise auf einem Mehrprozessorsystem mit einem gemeinsamen Hauptspeicher einsetzen: innerhalb eines Adreßraums können mehrere Threads existieren. Eine Realisierung dieses Modells erfordert einen Thread-Scheduler, sofern mehr Threads als Prozessoren vorliegen. Für den Spezialfall eines Einprozessorsystems ist ein Thread-Scheduling zwingend notwendig.

SM3 entspricht dem traditionellen Modell eines Mehrprozeßsystems vor der Einführung des Thread-Konzepts (z.B. UNIX): ein Thread-Wechsel führt automatisch zu einem Prozeß- und Adreßraumwechsel. Das Modell SM4 entsteht, wenn man für die Threads eines Prozesses ein eigenständiges Scheduling erlaubt. Dieser Ansatz findet beispielsweise Anwendung für die Verwaltung von User Level Threads, ist aber auch für das Scheduling von System Level Threads denkbar. Da das Thread-Scheduling prozeßspezifisch durchgeführt werden kann, ist für jeden Prozeß eine eigene Scheduling-Strategie einsetzbar. Tab. 5 enthält die Zuordnung zwischen den Scheduling-Modellen und den Thread-Scheduler- und Prozeß-Scheduler-Objekten.

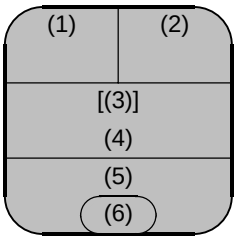
Modell	Prozeß	Thread	Prozeß-Scheduler	Thread-Scheduler
SM1	1	1	–	–
SM2	1	n (>1)	–	1
SM3	m (>1)	1	1	–
SM4	m (>1)	n (>1)	1	m

Tab. 5 Prozeß- und Thread-Scheduler innerhalb der Modelle SM1 bis SM4

9.3 Die Kontrollflüsse in den Modellen SM1 bis SM4

Die Modelle SM1 bis SM4 unterscheiden sich nicht nur bezüglich der benötigten Klassen (vgl. Tab. 5) sondern insbesondere auch in der Struktur der *Kontrollflüsse* bei ihren Operationen. Als Beispiel für die Analyse der auftretenden Kontrollflüsse wird die block-Operation auf einem Thread-Objekt verwendet: der auf dem Prozessor aktive Thread wechselt in den *Wartezustand*. Danach muß ein neuer Thread ausgewählt werden, der dem Prozessor zugeordnet wird (assign-Operation).

Die Kontrollflüsse zwischen den Objekten werden als *Zustandsautomaten* modelliert. Abb. 170 beschreibt die verwendete Notation. Die Angabe einer Objektreferenz in der Form [[x]] soll bedeuten, daß für diese Klasse nur genau ein Objekt existiert und die Referenz daher als bekannt angenommen werden kann. Die Abbildungen Abb. 171 (S. 318) bis Abb. 174 (S. 320) enthalten die Kontrollflüsse für die einzelnen Modelle. Ausgangspunkt ist die Aktivierung von `block`



(1) Eingabeparameter, die für diesen Zustand notwendig sind
(2) Parameter, die lediglich an andere Zustände weitergegeben werden
(3) Objektreferenz
(4) für das Objekt aufzurufende Operation
(5) Ausgabeparameter für Folgezustände
(6) Zustandsname

Abb. 170 Zustandsnotation

mit dem Parameter `event`. Dieser Parameter definiert das Ereignis, auf dessen Eintreffen der Thread wartet.

- Der Kontrollfluß der `block`-Operation im Modell SM1 (vgl. Abb. 171):
  Zunächst wird die `block`-Operation auf dem Thread-Objekt aufgerufen. Diese führt die lokale Zustandsänderung für das Thread-Objekt aus, z.B. das Sichern aller Register. Da es im Modell SM1 keine weiteren Prozesse oder Threads gibt, geht der Kontrollfluß in einen Wartezustand über (Zustand S2). Durch den Wartezustand übernimmt die darunter liegende Betriebssystemschicht die Kontrolle. Diese Schicht enthält dann unter anderem die Ereignisverwaltung, welche nach dem Auftreten des Ereignisses `event` den Kontrollfluß fortsetzt.[1] Als Ergebnis wird der Thread durch den Aufruf der `assign`-Operation wieder aktiviert.

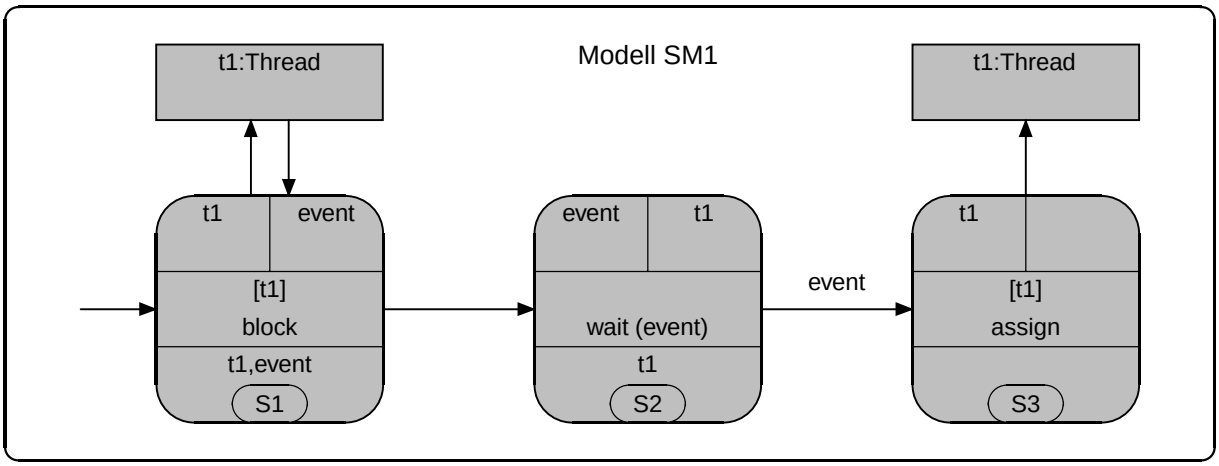


Abb. 171 Der Kontrollfluß der `block`-Operation für das Modell SM1

- Der Kontrollfluß der `block`-Operation im Modell SM2 (vgl. Abb. 172 (S. 319)):
  Da im SM2-Modell ein Prozeß mehrere Threads besitzen kann, wird der Kontrollfluß gegenüber SM1 um den Aufruf der `dispatch`-Operation auf dem ThreadScheduler-Objekt erweitert.
- Der Kontrollfluß der `block`-Operation im Modell SM3 (vgl. Abb. 173 (S. 319)):
  Im Modell SM3 wird ein ProcessScheduler-Objekt benötigt, da jetzt mehrere Prozesse vorliegen. ThreadScheduler-Objekte sind nicht erforderlich, da jeder Prozeß nur ein Thread-Objekt besitzt. Um ausgehend von einem Thread-Objekt das Process-Objekt zu ermitteln, wird die Relation R[P,T] verwendet, die die Zuordnung zwischen den Objekten der Klassen **Process** und **Thread** verwaltet.
- Der Kontrollfluß der `block`-Operation im Modell SM4 (vgl. Abb. 174 (S. 320)):
  In diesem Modell liegen mehrere Prozesse vor und jeder Prozeß besitzt ein ThreadScheduler-Objekt. Daher ist die Relation R[P,TS] erforderlich, um für eine Process-Referenz die zugehörige ThreadScheduler-Referenz zu bestimmen.

[1] In einem Betriebssystem wie SOLARIS entprächen die Thread-Objekte den *Light Weight Processes*, während die darunter liegende Ebene durch die *Kernel Threads* realisiert würde (vgl. [MM01, S. 15]).

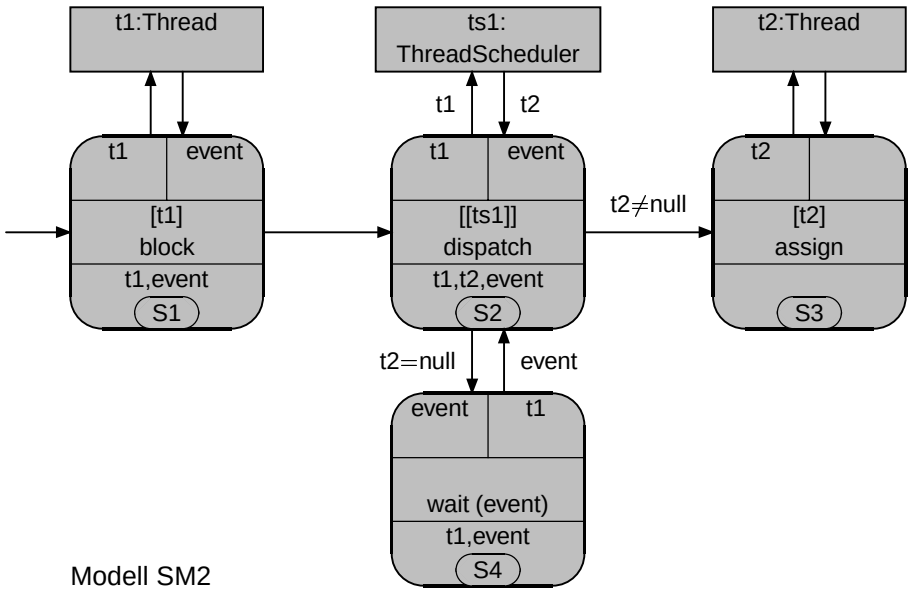


Abb. 172 Der Kontrollfluß der `block`-Operation für das Modell SM2

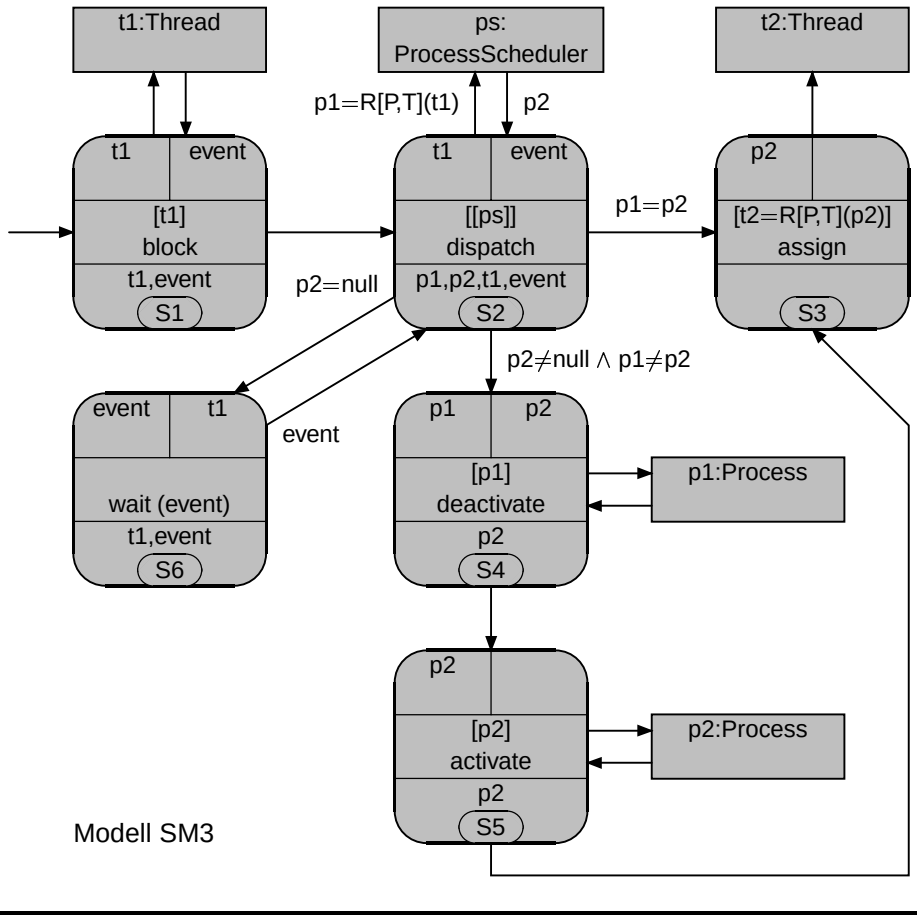


Abb. 173 Der Kontrollfluß der `block`-Operation für das Modell SM3

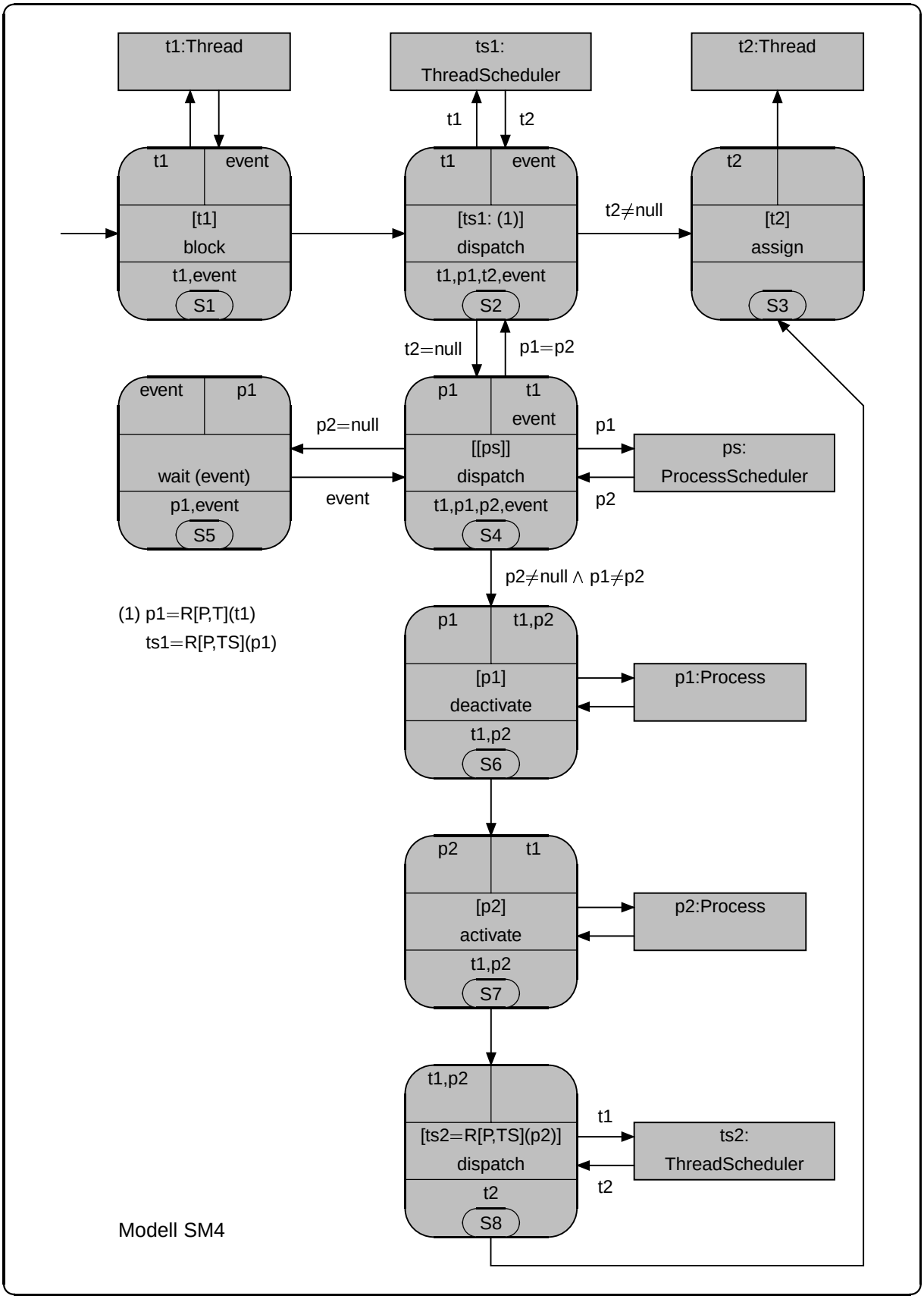


Abb. 174 Der Kontrollfluß der `block`-Operation für das Modell SM4

9.4 Die Erzeugung einer minimalen Scheduling-Komponente

9.4.1 Scheduling-Modelle aus Applikations- und Betriebssystemsicht

Unter der Voraussetzung, daß einer Applikation genau ein Prozeß zugeordnet wird, existieren applikationsbezogen nur die beiden Scheduling-Modelle SM1 und SM2: die Applikation ist intern *sequentiell* oder *nebenläufig* organisiert. Diese Modelle sind die ***Grundmodelle*** der Scheduling-Komponente.

Ein nicht adaptives Betriebssystem enthält üblicherweise ein statisch implementiertes Scheduling-Modell, das den allgemeinsten Fall abdeckt.[2] Zur Laufzeit auftretende Spezialfälle, die zu Strukturvereinfachungen der Scheduling-Komponente führen, können nicht ausgenutzt werden. Für die Realisierung einer minimalen Scheduling-Komponente sind folgende Informationen entscheidend:

- Welche Scheduling-Grundmodelle werden von den Applikationen benötigt?
  Aus der Sicht einer Applikation sind dies die Modelle SM1 oder SM2.
- Welches Scheduling-Modell entsteht aus der Kombination der (applikationsspezifischen) Grundmodelle?

Die Beantwortung der zweiten Frage erfordert eine Analyse der Kontrollflüsse, die bei den verschiedenen Kombinationen der Grundmodelle SM1 und SM2 auftreten. Die entstehenden Modelle werden als ***Ausführungsmodelle*** bezeichnet. Für die Charakterisierung der Ausführungsmodelle ist entscheidend, ob SM1 bzw. SM2 von *keiner*, *einer* oder *mehreren* Applikationen aktuell verwendet wird (siehe Tab. 6). Die Modelle SM5 bis SM8 müssen im Unterschied zu

SM1	1	–	n	–	n	1	n	1
SM2	–	1	–	m	m	1	1	m
entstehendes Modell	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8

Tab. 6 Kombinationen applikationsspezifischer Scheduling-Grundmodelle

SM1 bis SM4 berücksichtigen, daß Applikationen aus beiden Grundmodellen (SM1 und SM2) zu integrieren sind. Abb. 175 (S. 322) beschreibt beispielsweise den Kontrollfluß der block-Operation für das Modell SM5. Bei den Zuständen S1 und S7 hängt hier der Folgezustand von dem Grundmodell des zugehörigen Prozesses ab. In Abb. 175 (S. 322) sagt SM2(p1) beispielsweise aus, daß der Prozeß p1 dem Grundmodell SM2 zugeordnet ist.

9.4.2 Die dynamische Adaption der Scheduling-Komponente

Eine Adaption der Scheduling-Komponente kann immer dann notwendig werden, wenn eine neue Applikation *erzeugt* bzw. eine Applikation *beendet* wird.[3] Zu diesen Zeitpunkten muß

[2] Von den hier diskutierten Scheduling-Modellen ist dies SM4.

[3] Für die Scheduling-Komponente ist dies gleichbedeutend mit einer *Prozeßerzeugung* bzw. *Prozeßvernichtung*,

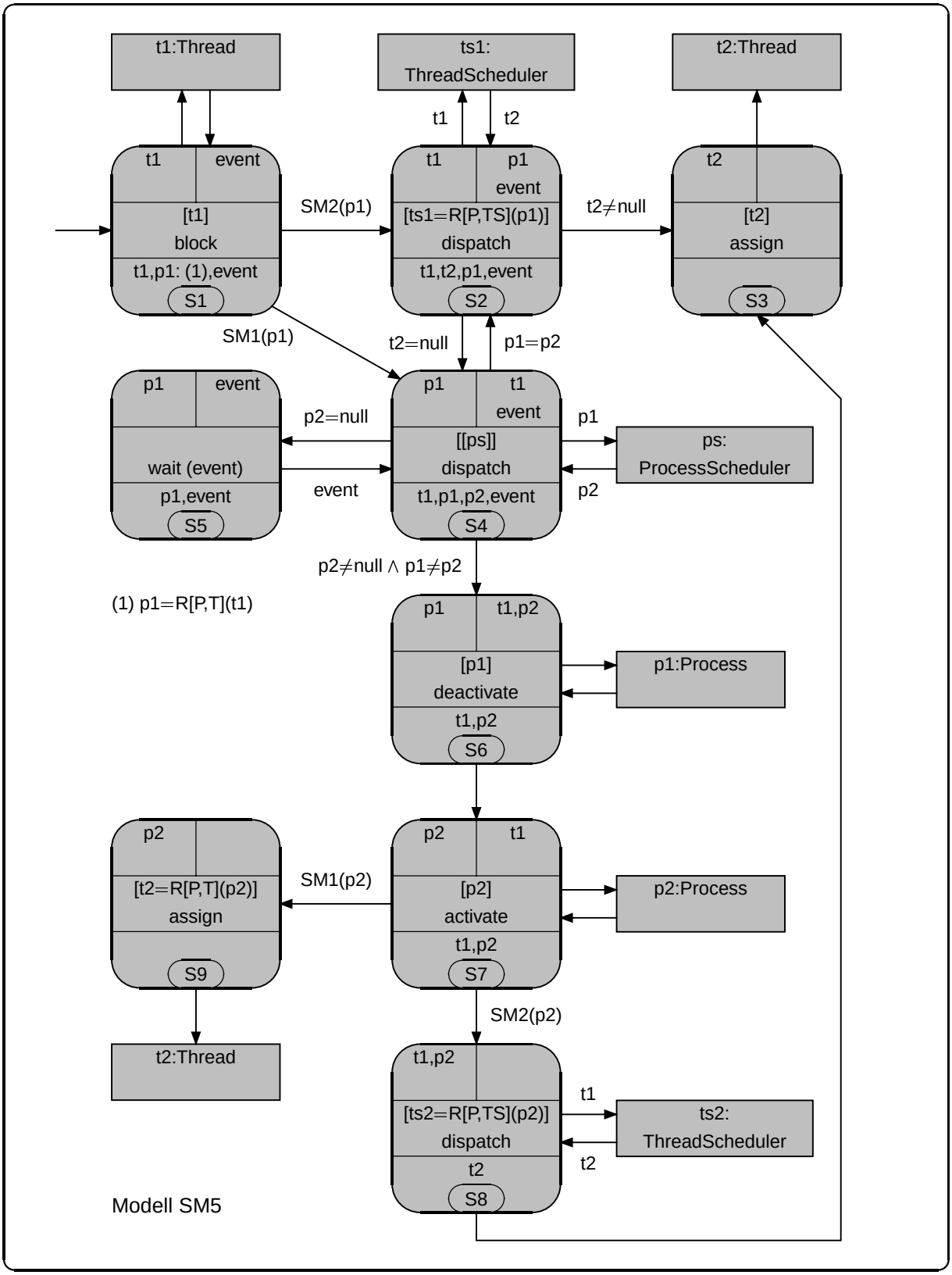


Abb. 175 Der Kontrollfluß der `block`-Operation für das Modell SM5

kontrolliert werden, ob ein Wechsel des Ausführungsmodells erforderlich ist. Tab. 7 (S. 323)

da eine 1-zu-1-Beziehung zwischen Applikationen und Prozessen vorausgesetzt wurde.

enthält die Beschreibung der Modellübergänge, wenn eine neue Applikation des Typs SM1 bzw. SM2 erzeugt wird. Liegt beispielsweise das Modell SM2 vor und eine SM1-Applikation kommt

kommt hinzu / liegt vor	—	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
SM1	SM1	SM3	SM6	SM3	SM8	SM5	SM7	SM7	SM5
SM2	SM2	SM6	SM4	SM7	SM4	SM5	SM8	SM5	SM8

Tab. 7 Modellübergänge beim Erzeugen einer Applikation

hinzu, so entsteht das neue, kombinierte Modell SM6. Die Modellübergänge beim Beenden einer Applikation sind in Tab. 8 dargestellt. Wenn der aktuelle Zustand über das Modell SM5

beendet / liegt vor	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
SM1	—		n>2: SM3 n=2: SM1		n>2: SM5 n=2: SM8	SM2	n>2: SM7 n=2: SM6	SM4
SM2		—		m>2: SM4 m=2: SM2	m>2: SM5 m=2: SM7	SM1	SM3	m>2: SM8 m=2: SM6

Tab. 8 Modellübergänge beim Beenden einer Applikation

definiert ist, gerade 2 Applikationen des SM2-Typs vorliegen und eine von ihnen beendet wird, dann findet ein Wechsel des Ausführungsmodells von SM5 nach SM7 statt.

Die Objekte der Klassen **Thread** und **Process** sind von dem Wechsel des Scheduling-Modells nicht betroffen. Dies ist eine wesentliche Voraussetzung für die Anwendung einer dynamischen Adaption, da die in den Thread- und Process-Objekten enthaltenen Zustandsinformationen bei einem Modellwechsel erhalten bleiben müssen. Für die einfache Austauschbarkeit der Ausführungsmodelle zur Laufzeit ist es entscheidend, daß die betroffenen Objekte keine Strukturinformationen enthalten, die von den globalen Kontrollflüssen benötigt werden. Daher wurden die Relationen R[P,T] und R[P,TS] eingeführt, die die Zuordnungen von Prozessen zu Threads und von Prozessen zu ihren Thread-Schedulern verwalten. Außerdem dürfen die globalen Kontrollflüsse bei einer Implementierung nicht auf die Objektoperationen *verteilt* werden, da sonst ein Kontrollflußaustausch nicht mehr möglich ist. Die Kontrollflüsse müssen folglich als eigenständige Objekte im System vorliegen.

9.5 Zielvorgaben für eine Erzeugung minimaler Betriebssystemkomponenten

Das Adaptionsmodell erlaubt eine Erzeugung minimaler Betriebssystemkomponenten für verschiedene Zielvorgaben. Durch die Auswahl des zur aktuellen Situation passenden Ausführungsmodells wird die Systemfunktionalität minimiert, was zu einer Optimierung der Ausführungszeiten der Operationsaufrufe führt. Falls eine Reduktion des Speicherplatzbedarfs im Vordergrund steht, werden nach einem Adaptionszyklus nicht mehr benötigte Objekte und Klassendefinitionen aus der Betriebssystemkomponente entfernt. Sind sehr häufige Ausführungsmodellwechsel notwendig, kann eine Reduktion des entstehenden Adaptionsaufwands erfolgen, indem (zumindest für eine bestimmte Zeit) auf die Rückkehr zu einfacheren Ausführungsmodellen verzichtet wird. Ist für eine Betriebssystemkomponente die Menge der benötigten Grundmodelle konstant, so läßt sich die Komponente auch statisch adaptieren. Für die identifizierten Grundmodelle wird zum Entwurfszeitpunkt das passende Ausführungsmodell ausgewählt und eine entsprechende minimale Betriebssystemkomponente erzeugt. Alle Teile der Komponenteninfrastruktur, die nur für den dynamischen Adaptionsprozeß erforderlich sind, können somit entfallen.

9.6 Die Umsetzung der Scheduling-Komponente durch ein Variantenmodell

Für die Realisierung der Scheduling-Komponente wird ein Variantenmodell mit der folgenden Struktur gewählt:

- Die Klasse **Scheduling** ist die Chassis-Klasse.
- Um die Prozesse den Grundmodellen SM1 bzw. SM2 zuordnen zu können, werden zwei Prozeßklassen **ProcessSM1** und **ProcessSM2** definiert, die als Component-Klassen dienen. Es wird davon ausgegangen, daß die Scheduling-Komponente jeweils maximal 128 Prozesse der beiden Typen verwalten kann.
- Für die zwei Grundmodelle werden die Thread-Klassen **ThreadSM1** und **ThreadSM2** zur Verfügung gestellt. Von jedem Thread-Typ sind maximal 1024 Objekte vorhanden.
- Die Klassen **ProcessScheduler** und **ThreadScheduler** sind ebenfalls Component-Klassen von **Scheduling**. In die Scheduling-Komponente können ein **ProcessScheduler**-Objekt und maximal 128 **ThreadScheduler**-Objekte integriert werden. Damit steht für jeden SM2-Prozeß ein **ThreadScheduler**-Objekt zur Verfügung.

Die korrespondierenden Klassen des Variantenmodells der Scheduling-Komponente sind in Abb. 176 (S. 325) dargestellt. Dabei sind nur diejenigen Operationen angegeben, die für die Umsetzung der globalen Kontrollflüsse der `block`-Operation eines Thread-Objekts notwendig sind. Die korrespondierende Klasse **SchedulingDsc** realisiert intern die Relationen R[P,T] und R[P,TS]. Für die Verwaltung der Relationen werden die `connect-` und `disconnect`-Operationen zur Verfügung gestellt. Wird beispielsweise ein neuer SM1-Thread für einen SM1-Prozeß erzeugt, dann ist die `connect`-Operation für diese beiden Objekte auszuführen, bevor das Thread-Objekt über die `installThreadSM1`-Operation in das Scheduling-Objekt

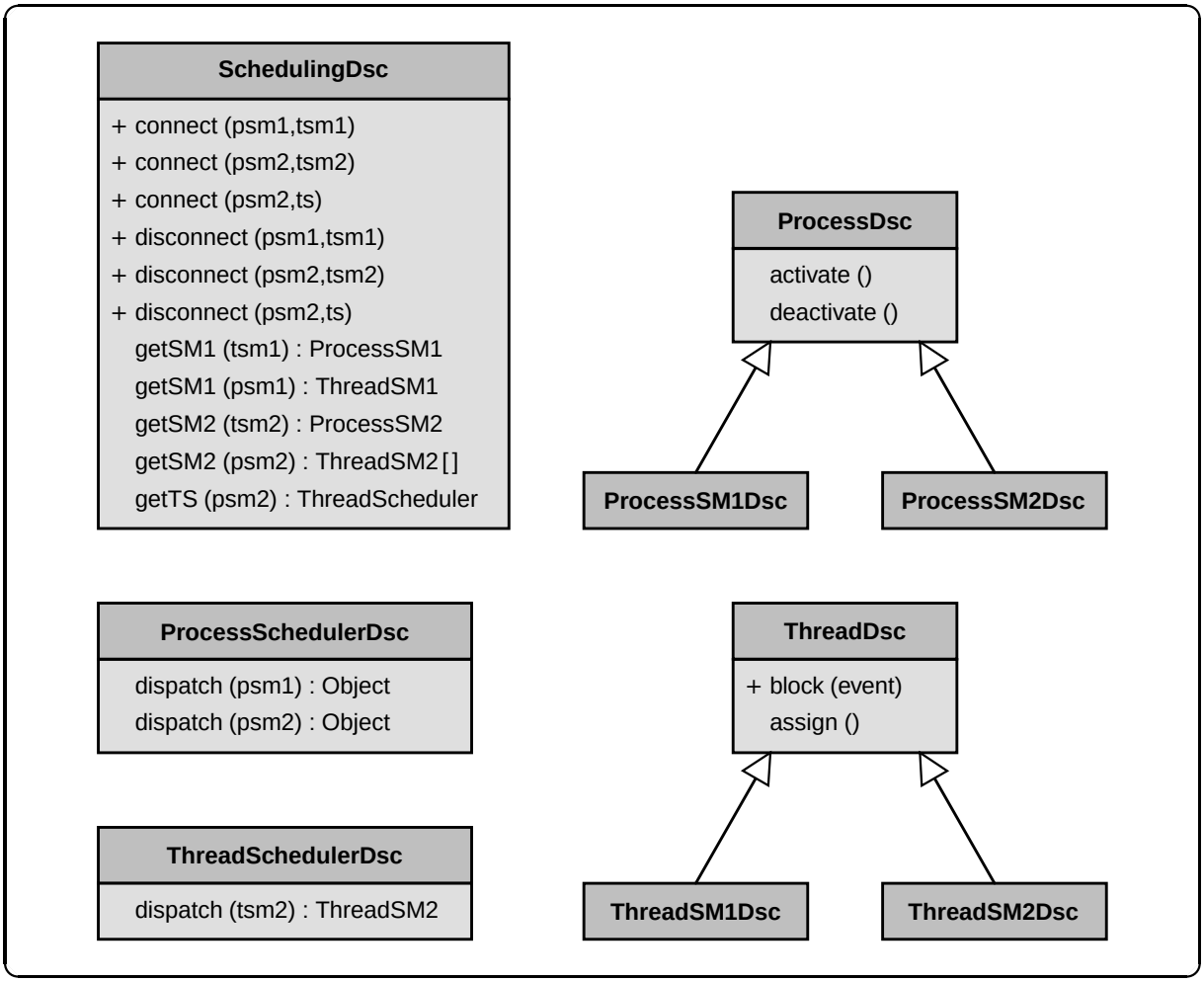


Abb. 176 Die korrespondierenden Klassen der Scheduling-Komponente

eingebaut wird.

Die resultierende Struktur des statischen Modells der Scheduling-Komponente zeigt Abb. 177 (S. 326). Der Adaptionsprozeß findet statt, wenn ein Prozeßobjekt durch eine `install`-Operation integriert oder durch eine `remove`-Operation entfernt wird. Hierfür stehen die Script-Klassen **SchedulingThreadBlockSM1** bis **SchedulingThreadBlockSM8** zur Verfügung. Diese implementieren die Zustandsautomaten für die globalen Kontrollflüsse der `block`-Operation. Das dynamische Modell der Scheduling-Komponente ist in Abb. 178 (S. 327) dargestellt. Aus Platzgründen sind nur für die Script-Klasse **SchedulingThreadBlockSM1** Assoziationen angegeben. Die benötigten Assoziationen der anderen Script-Klassen ergeben sich aus den zugehörigen Zustandsautomaten. Für die Klasse **SchedulingThreadBlockSM5** sind dies beispielsweise Assoziationen zu den Klassen **ThreadSM2**, **ProcessSM2**, **ThreadScheduler** und **ProcessScheduler**. Das Konfigurationswissen für den Adaptionsprozeß ist in dem Objekt der Klasse **SchedulingConfiguration** enthalten. Wird z.B. ein neues **ProcessSM2**-Objekt in die Scheduling-Komponente integriert, so überprüft die `installProcessSM2`-Operation des Konfigurationsobjekts, ob dadurch ein Wechsel des Ausführungsmodells erforderlich ist. Wenn dies der Fall ist, werden die Script-Objekte für die `block`-Operation bei allen Thread-Objekten ausgetauscht, wodurch die neuen globalen Kontrollflüsse installiert sind.

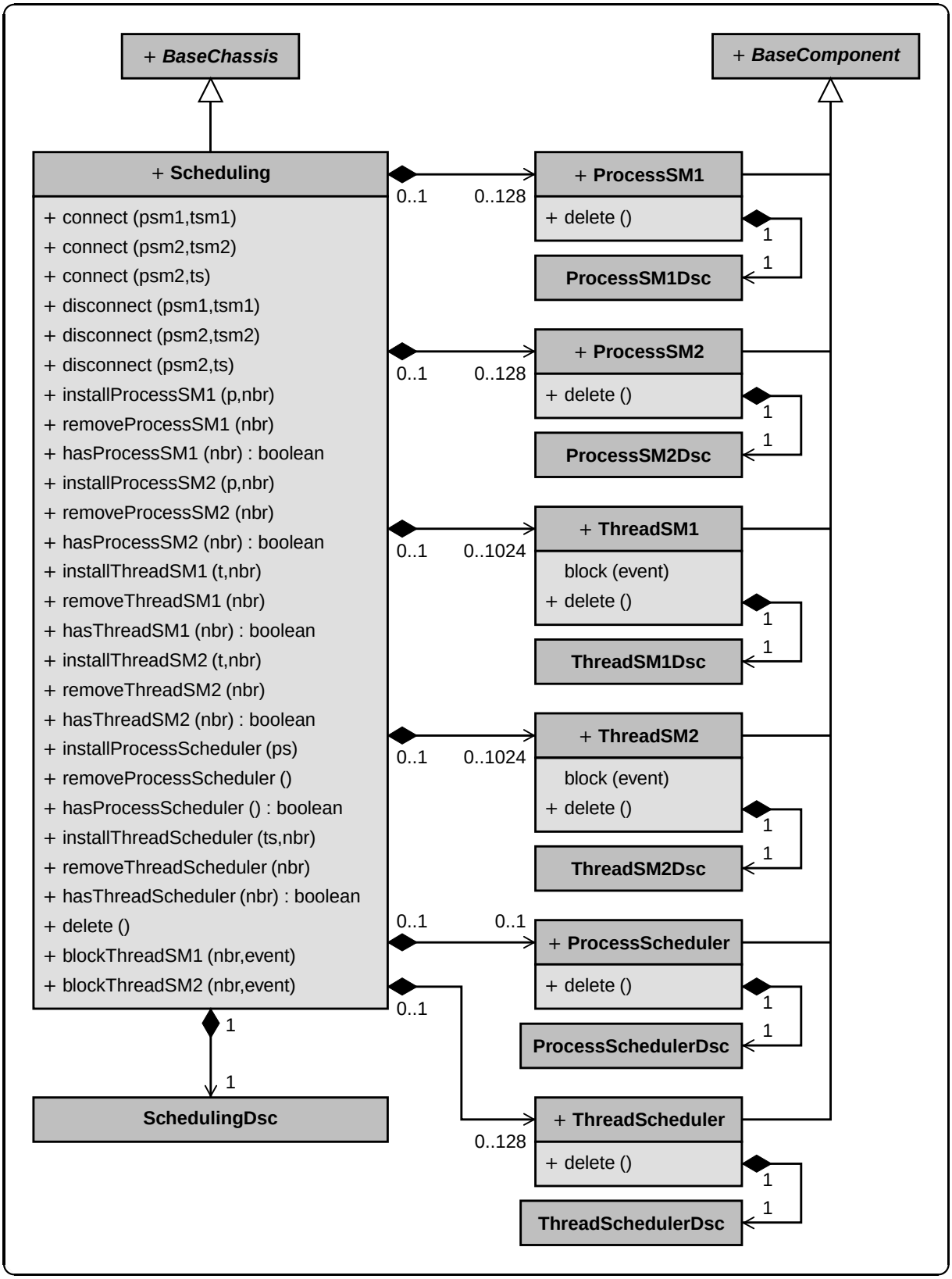


Abb. 177 Das statische Modell der Scheduling-Komponente

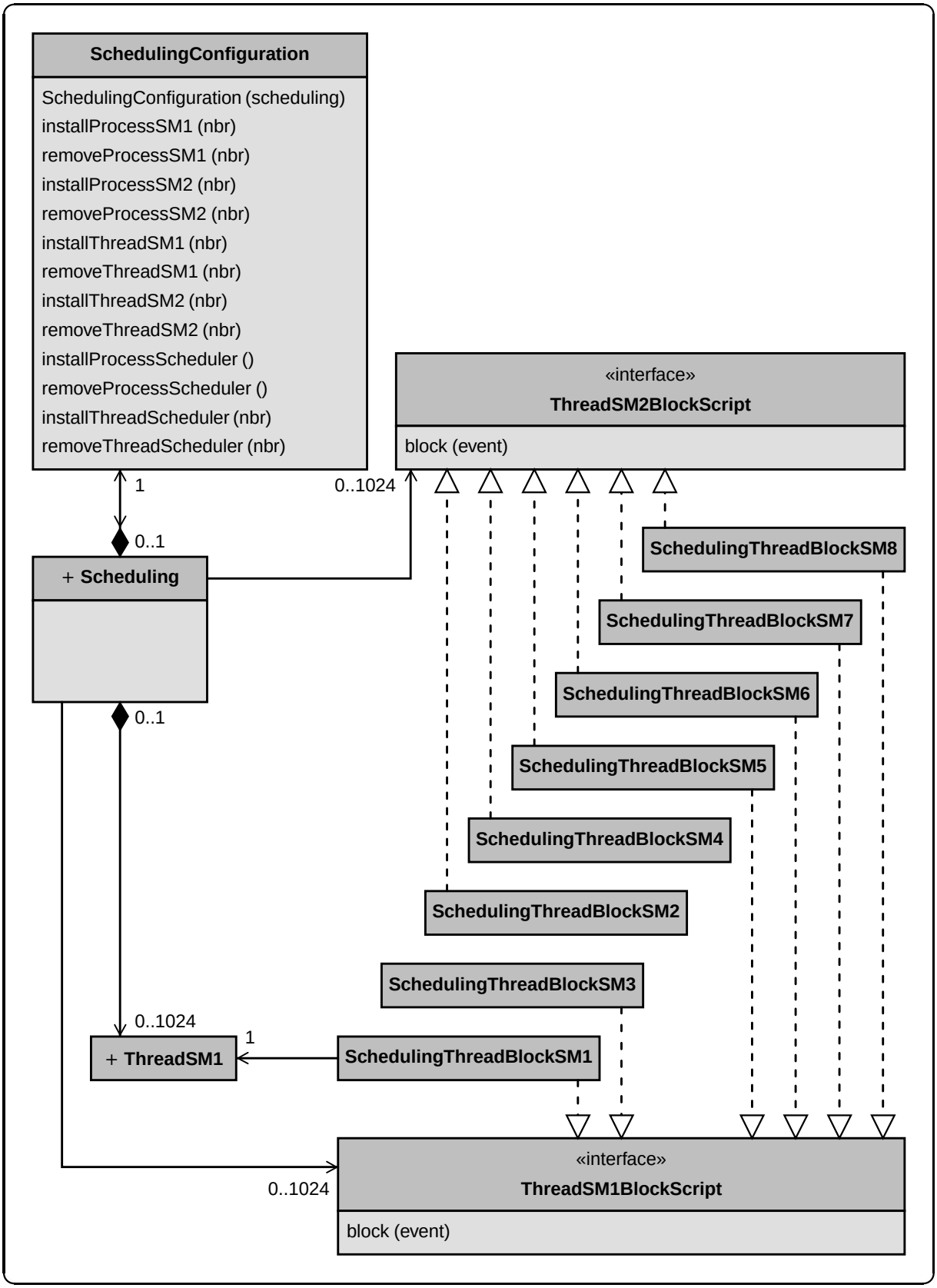


Abb. 178 Das dynamische Modell der Scheduling-Komponente

9.7 Kapitelzusammenfassung

In diesem Kapitel wurde zunächst das Konzept eines dynamisch adaptiven Betriebssystems an dem Beispiel der Scheduling-Komponente vorgestellt. Ausgehend von vier Scheduling-Modellen wurden die Grund- und Ausführungsmodelle einer Scheduling-Komponente definiert. Für diese Modelle wurden die globalen Kontrollflüsse analysiert, die beim Aufruf einer block-Operation für ein Thread-Objekt vorliegen.

Bei der Erzeugung einer Applikation wird von dieser ein Grundmodell gewählt. Aus den aktuell vorhandenen Grundmodellen ergibt sich das Ausführungsmodell der Scheduling-Komponente. Kommt es durch das Erzeugen oder Beenden einer Applikation zu einem Ausführungsmodellwechsel, findet eine Adaption der Scheduling-Komponente statt. Um den Adaptionsprozeß durchführen zu können, ist es entscheidend, die globalen Kontrollflüsse in eigenen Objekten zu kapseln.

Für die Scheduling-Komponente wurde dann ein Variantenmodell erstellt. Die globalen Kontrollflüsse wurden durch entsprechende Script-Klassen realisiert. Das Konfigurationswissen für den Adaptionsprozeß wurde dem SchedulingConfiguration-Objekt zugeordnet, dessen install- und remove-Operationen dann auch die Adaption durch den Austausch der Script-Objekte vornehmen.

Insgesamt zeigte sich, daß das Konzept einer minimalen, applikationsspezifischen Scheduling-Komponente über ein Variantenmodell beschrieben und umgesetzt werden kann.

Kapitel 10

Zusammenfassung und Ausblick

Die Zielsetzung der Arbeit war es, für zwei wichtige Modellierungsbereiche, die Rollen- und die Variantenmodellierung, eine möglichst weitgehende Integration der Software-Entwicklungsphasen Analyse, Entwurf und Implementierung zu erreichen.

Hierfür wurde in Kapitel 2 zunächst auf den Software-Entwicklungsprozeß eingegangen. Nach einer kurzen Vorstellung der Phasen Analyse, Entwurf, Implementierung und Test wurden auf dieser Grundlage Prozeßmodelle zur Software-Entwicklung beschrieben. Den Schwerpunkt bildete hierbei der in dieser Arbeit eingesetzte Balancierte Makroprozeß. Dieser Prozeß wurde einerseits als Software-Entwicklungsprozeß angenommen, den ein Anwendungsentwickler zur Realisierung von Variantenmodellen benutzt. Zum anderen bildete er die Grundlage für ein iteratives und evolutionäres Vorgehensmodell zur Analyse und Beschreibung der Funktionalität der entwickelten neuen Rollen- und Variantenmodelle. Des weiteren wurde die heute bei Software-Systemen sehr häufig verwendete Drei-Schichten-Architektur vorgestellt, die später bei der Beschreibung der Integration der Rollen- und Variantenmodelle in den Systementwicklungsprozeß aus der Anwendersicht als Referenzarchitektur diente.

Im dritten Kapitel wurde an zwei Beispielen gezeigt, daß ein Rollenmodell sehr gut zur Modellierung von Objekten geeignet ist, die im Laufe ihres Lebenszyklus Strukturänderungen unterliegen. Das Ergebnis war, daß die objektbasierte Vererbung einen wichtigen Ausgangspunkt für die Realisierung von Rollenkonzepten darstellt, allein aber nicht ausreicht, um allen Anforderungen an ein flexibles Rollenkonzept gerecht zu werden. Als wichtige Anforderungen wurden identifiziert:

- Das neue Rollenkonzept sollte die Formulierung von Rollenmodellen erlauben, deren prinzipielle Struktur über einen azyklischen Graphen beschreibbar ist.
- Für die Spezifikation, unter welchen Bedingungen Rollen übernommen werden dürfen, mußten auf der Analyseebene geeignete Restriktionen angeboten werden.
- Das Rollenmodell sollte vollständig in den Software-Entwicklungsprozeß integriert werden. Hierbei war die Konsistenz der Repräsentationen des Rollenmodells auf der Analyse-, Entwurfs- und Implementierungsebene sicherzustellen. Um diese Anforderung zu erfüllen, sollte ein Generierungsprozeß verwendet werden, der ausgehend von der abstrakten Beschreibung des Rollenmodells auf der Analyseebene automatisch die innerhalb der Implementierungsebene benötigten Strukturen erzeugt.

In Kapitel 4 wurden alternative Ansätze zur Modellierung und Realisierung von Rollenkonzepten vorgestellt. Diese Ansätze ließen sich in zwei große Klassen untergliedern: Rollenkonzepte zur Modellierung des dynamischen Vererbungscharakters und Rollenkonzepte für die Modellierung von Objektkollaborationen. Für die insgesamt 24 diskutierten Ansätze wurde am Ende des Kapitels eine vergleichende Bewertung anhand von acht Kriterien vorgenommen:

- Welche Struktur besitzt das Modell (Baum oder Graph)?
- Sind Mehrfachrollen erlaubt?
- Können Rollen dynamisch zur Laufzeit übernommen und wieder abgegeben werden?
- Wird für das Gesamtobjekt, d.h. das Objekt mit allen seinen aktuellen Rollen, ein Identitätsbegriff unterstützt?
- In welcher Form ist eine Redefinition von Methoden möglich?
- Lassen sich für das Rollenmodell Restriktionen spezifizieren?
- Wird eine Rollenmigration unterstützt?
- Liegt Typsicherheit vor?

Es stellte sich heraus, daß keiner der alternativen Ansätze gleichzeitig die folgenden, wünschenswerten Eigenschaften besitzt:

- Als Struktur wird ein azyklischer Graph unterstützt.
- Von einem Rollentyp können mehrere Rollenobjekte existieren.
- Es lassen sich auf der Modellierungsebene Restriktionen spezifizieren, die automatisch in eine Implementierung integriert werden.
- Es liegt eine vollständige Typsicherheit vor.

In Kapitel 5 wurde ein neues Rollenkonzept zur Realisierung von Rollenmodellen mit Mehrfachvererbung entwickelt. Das Rollenkonzept erfüllt alle geforderten Eigenschaften und besitzt als zentrale Eigenschaft eine klare Trennung zwischen der potentiellen Struktur eines Gesamtobjekts und der Spezifikation der Funktionalität der einzelnen Rollen.

Für die Beschreibung einer Rollenmodellstruktur wurde eine Grammatik definiert, um die folgenden Eigenschaften festzulegen:

- Welche direkten Unterrollen darf eine Rolle besitzen?
  Auf diesem Weg läßt sich für ein Rollenmodell ein azyklischer Graph definieren, der mehrere Wurzelrollen enthält.
- Wieviele Rollenobjekte dürfen von einer Unterrolle gleichzeitig existieren?
- Welche Rollen sind abstrakt?
- Zwischen welchen Rollen bestehen welche Restriktionen?
  Als Restriktionen stehen die **and**-Restriktion, die **or**-Restriktion, die **exor**-Restriktion und die **nocreation**-Restriktion zur Verfügung. Die ersten beiden Restriktionen werden implizit über die Struktur des Rollenmodells definiert, während die letzten beiden Restriktionen explizit vom Anwender anzugeben sind.

Die Strukturbeschreibung dient zusammen mit der Spezifikation der Funktionalität der Rollen über ihre korrespondierenden Klassen als Grundlage für den Generierungsprozeß. Die erzeugten Rollenklassen können später direkt von der Applikation verwendet werden.

Um die Konsistenz zwischen der Modellierungs- und der Implementierungsebene sicherzustellen, mußten verschiedene Testkriterien entwickelt werden. Die Tests sind entscheidend für die Frage, ob eine Unterrolle erzeugt werden darf:

- Existieren alle benötigten direkten Oberrollenobjekte?
- Kann durch die Erzeugung der Unterrolle mit den angegebenen direkten Oberrollenobjekten die vorgegebene Struktur des Rollenmodells verletzt werden?
- Ist aufgrund der spezifizierten Kardinalitäten die Unterollenerzeugung überhaupt noch erlaubt?
- Werden Restriktionen verletzt?

Die Ermittlung der konkreten Testkriterien ist für komplexe Rollenmodelle teilweise sehr aufwendig und damit fehleranfällig. Daher war es wichtig, diese Tests direkt in die erzeugten Rollenklassen zu integrieren.

Weitere wichtige Eigenschaften des entwickelten Rollenmodells sind:

- Die Trennung zwischen der Struktur- und der Funktionalitätsdefinition entspricht dem Konzept der aspektorientierten Programmierung. Die neu definierte Beschreibungssprache kann als Aspektbeschreibungssprache aufgefaßt werden. Durch sie werden alle Informationen beschrieben, die den *Aspekt der Rollenmodellstruktur* betreffen.
- Die korrespondierenden Klassen können in unterschiedlichen Rollenmodellen wiederverwendet werden, da sie selbst keine Informationen darüber enthalten, welche Stellung sie in einem Rollenmodell haben.
- Durch die für die einzelnen Rollenklassen erzeugten Verwaltungsoperationen wird eine Gesamtobjektidentität gewährleistet. Obwohl intern das Gesamtobjekt aus mehreren Einzelobjekten besteht, tritt das Problem der Objektschizophrenie nicht auf, da die Verwaltungsoperationen keine Inkonsistenzen des Gesamtobjekts zulassen. Für den Anwender wird damit immer eine konsistente Gesamtobjektsicht garantiert.
- Es wird eine vollständige Typsicherheit erreicht.
- Für die Verwaltungsoperationen wurden entsprechende Exception-Klassen definiert. Dadurch wird der Anwender gezwungen, explizit auf Fehler zu reagieren, was die Software-Qualität erhöht.
- Unter der Annahme, daß für die korrespondierenden Klassen eine vollständige Kapselung vorliegt, ist es ohne großen Aufwand möglich, in dem Modell den Austausch der korrespondierenden Objekte durch eine neue Implementierung zur Laufzeit zu unterstützen.

Nach der Vorstellung der Eigenschaften des neuen Rollenmodells wurde beschrieben, wie sich dieses aus der Sicht eines Anwendungsentwicklers in den übergeordneten Software-Entwicklungsprozeß integrieren läßt. Den Abschluß des Kapitels stellte die Beschreibung der Software-Architektur zur Generierung des Rollenmodells dar, die in JAVA realisiert wurde.

Im sechsten Kapitel wurde zunächst eine allgemeine Abgrenzung von Rollen- und Variantenmodellen vorgenommen. Danach wurden die Anforderungen an ein Variantenmodell ermittelt:

- AV1: Es muß die hierarchische Konstruktion von Bauteilen unterstützt werden.
  In ein Grundobjekt (Chassis) können unterschiedliche Komponenten eingebaut und eventuell auch wieder ausgebaut werden.

- AV2: Eine Komponente, die in ein Chassis einbaubar ist, kann selbst wieder aus untergeordneten Bauteilen bestehen, d.h. ebenfalls die Chassis-Funktionalität übernehmen.
- AV3: Eine Komponente kann physikalisch maximal Bestandteil eines Chassis sein.
- AV4: Die aktuell verfügbare Funktionalität (d.h. die Menge der nutzbaren Operationen) einer Komponente kann davon abhängen, in welches Chassis sie eingebaut ist.
- AV5: Die Funktionalität einer Chassis-Operation kann von dem Vorhandensein bestimmter Komponenten oder Komponentenkombinationen abhängen.
- AV6: Die Funktionalität einer Komponentenoperation kann von dem Vorhandensein anderer Komponenten oder Komponentenkombinationen abhängen.

Zusätzlich wurde wieder, wie bei den Rollenmodellen, die Anforderung nach einer möglichst vollständigen Integration in den Software-Entwicklungsprozeß gestellt.

In Kapitel 7 wurden alternative Ansätze zur Unterstützung der Variantenbildung betrachtet. Die zugehörigen Konzepte sind eng mit Kompositionstechniken verbunden, die sich in zwei Kategorien unterteilen lassen, je nachdem, ob sie auf der *ist-ein*- oder der *besteht-aus*-Beziehung aufbauen. Aus beiden Bereichen wurden insgesamt acht Ansätze untersucht. Es stellte sich heraus, daß alle Ansätze prinzipiell geeignet sind, AV1 bis AV6 zu erfüllen, allerdings nur mit einem erheblichen Aufwand, da kein Ansatz eine explizite Unterstützung aller Anforderungen anbietet. Bei den Ansätzen aus dem Vererbungsumfeld kam außerdem die Einschränkung hinzu, daß sie die Variantenbildung einer Klasse über eine *ist-ein*-Beziehung realisieren und so aus konzeptueller Sicht nicht zu dem Variantenmodell aus Kapitel 6 passen.

Kapitel 8 stellte das neu entwickelte Variantenmodell vor. Dieses erfüllt die Anforderungen durch die Trennung der Aspekte *Strukturbeschreibung*, *Objektfunktionalität*, *globaler Kontrollfluß* und *Konfigurationswissen*.

Für die Beschreibung der Struktur eines Variantenmodells wurde eine Grammatik definiert, über die sich die Anforderungen AV1, AV2 und AV4 umsetzen lassen. Da die Grammatik die Spezifikation von Variantenmodellen mit einer beliebigen Hierarchiestufenanzahl erlaubt, sind die Anforderungen AV1 und AV2 erfüllt. Zusätzlich kann über die Strukturbeschreibung spezifiziert werden, welche Operationen an der Schnittstelle einer Chassis- bzw. Chassis-Komponentenklasse sichtbar sind. Damit ist die Anforderung AV4 umgesetzt. Die Spezifikation der Objektfunktionalität erfolgt über eine korrespondierende Klasse. Hier werden die Operationen definiert und implementiert, die den internen Zustand einer Komponente betreffen. Die korrespondierenden Klassen bilden zusammen mit der Strukturbeschreibung den Ausgangspunkt für die Generierung des Variantenmodells. Zu jeder korrespondierenden Klasse wird eine Zugriffsklasse erzeugt. Die Chassis- und Chassis-Komponentenzugriffsklassen besitzen entsprechende install- und remove-Operationen für den Ein- und Ausbau der Komponentenobjekte. Um die Anforderung AV3 zu erfüllen, daß eine Komponente physikalisch maximal Bestandteil eines Chassis sein kann, wurden Zustandsautomaten für Chassis-, Chassis-Komponenten und Komponentenklassen definiert, deren Funktionalität bei der Generierung in die entsprechenden Klassen integriert wird. Diese Zustandsautomaten überwachen dann den Ein- und Ausbau von Komponenten. Die Anforderungen AV5 und AV6 betrafen die globalen Kontrollflüsse zwischen den beteiligten Objekten und deren Konfigurierbarkeit. Um die globalen Kontrollflüsse in einem konventionellen Ansatz von der aktuellen Konfiguration abhängig zu machen, wird man typi-

scherweise `if`- und `switch`-Anweisungen innerhalb der vom globalen Kontrollfluß betroffenen Operationen verwenden. Dies entspricht einer Verteilung des Konfigurationswissens über die einzelnen Objekte. Damit werden die globalen Kontrollflüsse schwer nachvollziehbar und es ist sehr aufwendig, Änderungen vorzunehmen. Aus diesem Grund wurde in dem entwickelten Modell das Konfigurationswissen in einer eigenen Konfigurationsklasse konzentriert. Jedes Chassis-Objekt besitzt dann ein eigenes Konfigurationsobjekt. Dieses stellt die Funktionalität der install- und remove-Operationen zur Verfügung, und legt damit fest, welche Kontrollflüsse durch eine Konfigurationsänderung betroffen sind. Ebenso wie das Konfigurationswissen wurden die globalen Kontrollflüsse in eigenen Objekten gekapselt, den Script-Objekten. Diese können als Implementierungen von Sequenzdiagrammen angesehen werden, da sie zum einen die Referenzen auf die betroffenen Objekte enthalten und zum anderen die Operationsaufrufe.

Weitere wichtige Eigenschaften des vorgestellten Ansatzes sind:

- Die spezifizierte Beschreibungssprache läßt sich als Aspektbeschreibungssprache auffassen, die den *Aspekt Variantenmodellstruktur* betrifft.
- Die korrespondierenden Klassen können in unterschiedlichen Variantenmodellen wiederverwendet werden, da sie selbst keine expliziten Beziehungen zu (anderen) korrespondierenden Klassen besitzen. Auf diesem Weg ist es sehr einfach, unterschiedliche Ausprägungen eines Variantenmodells zu erzeugen.
- Durch die Generierung spezieller Verwaltungsoperationen für die beteiligten Klassen und die Verwendung der Sprache JAVA wird eine vollständige Typsicherheit erreicht.
- Die Verwaltungsoperationen erzeugen im Fehlerfall entsprechende Exception-Objekte. Damit wird der Anwendungsentwickler gezwungen, auf Fehler zu reagieren, was zu einer Steigerung der Software-Qualität führt.

Nach der Vorstellung des Analyse- und des Entwurfsmodells für die Umsetzung des neuen Variantenmodells wurde die Software-Architektur zur Generierung des Variantenmodells beschrieben, die wesentliche Teile der Software-Architektur des Rollenmodellgenerators wiederverwendet. Den Abschluß des Kapitels stellte die Integration des Variantenmodells in den Software-Entwicklungsprozeß dar. Hier wurden die aus der Sicht des Anwendungsentwicklers notwendigen Schritte zur Realisierung eines Variantenmodells zusammengefaßt.

In Kapitel 9 wurde eine Fallstudie zur Realisierung der Scheduling-Komponente eines dynamisch adaptierbaren Betriebssystems vorgestellt. Der erste Schritt war die Beschreibung des Konzepts eines dynamisch adaptiven Betriebssystems an dem Beispiel der Scheduling-Komponente. Ausgehend von vier Scheduling-Modellen wurden die Grund- und Ausführungsmodelle einer Scheduling-Komponente definiert. Für diese Modelle wurden die globalen Kontrollflüsse analysiert, die beim Aufruf einer block-Operation für ein Thread-Objekt vorliegen.

In einem zweiten Schritt wurde für die Scheduling-Komponente dann ein Variantenmodell erstellt. Die globalen Kontrollflüsse wurden durch entsprechende Script-Klassen realisiert. Das Konfigurationswissen für den Adaptionsprozeß wurde dem SchedulingConfiguration-Objekt zugeordnet, dessen install- und remove-Operationen die Adaption durch den Austausch der betroffenen Script-Objekte vornehmen.

Insgesamt zeigte sich, daß das Konzept einer minimalen, applikationsspezifischen Scheduling-Komponente über ein Variantenmodell beschrieben und umgesetzt werden kann.

Interessante Ansatzpunkte für eine Weiterentwicklung der vorgestellten Konzepte betreffen zum einem die Funktionalitätserweiterung der Rollen- und Variantenmodelle und eine Integration der beiden Konzepte.

Als eine Ergänzung des Rollenmodellkonzepts sind folgende Bereiche denkbar:

- Die Spezifikation zusätzlicher Restriktionen, um eine noch direktere Abbildung der Realität in ein Rollenmodell zu erreichen.
- Die Bereitstellung weiterer Verwaltungsoperationen innerhalb der erzeugten Rollenklassen, um den Anwendungen eine komfortablere Schnittstelle anzubieten.
- Die Migration von Rollenobjekten.
- Die Umsetzung von Objektkollaborationen.
- Der Austausch der korrespondierenden Objekte zur Laufzeit durch eine neue Version, um Implementierungsfehler zu beheben oder eine effizientere Implementierung bereitzustellen.
- Die Entwicklung einer grafischen Benutzungsschnittstelle zur Spezifikation der Rollenmodelle.

Eine Weiterentwicklung der Variantenmodelle kann an den folgenden Punkten ansetzen:

- Der Austausch des Konfigurationswissens zur Laufzeit.
- Der Ein- und Ausbau neuartiger Komponenten zur Laufzeit, d.h. Komponenten, die in der aktuellen Version des Variantenmodells noch nicht bekannt waren.
- Die Formulierung von Strukturrestriktionen, z.B. *wenn ein Bauteil des Typs X bereits eingebaut ist, kann ein Bauteil des Typs Y nicht mehr installiert werden*.
- Der Austausch der korrespondierenden Objekte zur Laufzeit durch eine neue Version.
- Die Entwicklung einer grafischen Benutzungsschnittstelle zur Spezifikation der Variantenmodelle.

Ein weiteres interessantes Arbeitsgebiet stellt die Integration der vorgestellten Ansätze dar. Ein Objekt wird zunächst über ein Variantenmodell beschrieben, und kann dann zusätzlich unterschiedliche Rollen annehmen. Da die entwickelten Software-Architekturen für die Generierung der Rollen- und Variantenmodelle sehr große Gemeinsamkeiten aufweisen, ist es als längerfristiges Forschungsziel auch vorstellbar, eine Software-Architekur zur automatischen Erzeugung von Modellgeneratoren zu entwickeln.

Literaturverzeichnis

[AAG00] ALBANO, ANTONIO, GIUSEPPE ANTOGNONI und GIORGIO GHELLI: *View Operations on Objects with Roles for a Statically Typed Database Language*. IEEE Transactions on Knowledge and Data Engineering, 12(4):548–567, 2000.

[ABGO93] ALBANO, A., R. BERGAMINI, G. GHELLI und R. ORSINI: *An Object Data Model With Roles*. In: *Proceedings of the Nineteenth International Conference on Very Large Data Bases (VLDB)*, Seiten 39–51. Morgan Kaufmann, 1993.

[ACO85] ALBANO, A., L. CARDELLI und R. ORSINI: *Galileo: A Strongly Typed, Interactive Conceptual Language*. ACM Transactions on Database Systems, 10(2):230–260, 1985.

[ADG95] ALBANO, ANTONIO, MILENA DIOTAVELLI und GIORGIO GHELLI: *Extensible Objects for Database Evolution: Language Features and Implementation Issues*. In: ATZENI, PAOLO und VAL TANNEN (Herausgeber): *Proceedings of the Fifth International Workshop on Database Programming Languages (DBPL-5)*, Electronic Workshops in Computing. Springer-Verlag, 1995.

[AGH00] ARNOLD, KEN, JAMES GOSLING und DAVID HOLMES: *The Java Programming Language*. Addison Wesley Publishing Company, 3. Auflage, 2000.
ISBN 0-201-70433-1.

[AGO95] ALBANO, A., G. GHELLI und R. ORSINI: *Fibonacci: A Programming Language for Object Databases*. The VLDB Journal, 4(3):403–439, 1995.

[ALZ00] ANCONA, DAVIDE, GIOVANNI LAGORIO und ELENA ZUCCA: *Jam – A Smooth Extension of Java with Mixins*. In: BERTINO, ELISA (Herausgeber): *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '00)*, LNCS 1850, Seiten 154–178. Springer-Verlag, 2000.

[And97] ANDERSEN, EGIL P.: *Conceptual Modeling of Objects: A Role Modeling Approach*. Doktorarbeit, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 1997.
ftp://ftp.nr.no/pub/egil/ConceptualModelingOO.ps.gz.

[And03] ANDRESEN, ANDREAS: *Komponentenbasierte Softwareentwicklung mit MDA, UML und XML*. Hanser, 2003. ISBN 3-446-22282-0.

[ANM+90] ADER, MARTIN, OSCAR NIERSTRASZ, STEPHEN MCMAHON, GERHARD MÜLLER und ANNA-KRISTIN PRÖFROCK: *The ITHACA Technology: A Landscape for Object-Oriented Application Developement*. In: *Proceedings, Esprit 1990 Conference*, Seiten 31–51, Monticello, Illinois, USA, 1990. Kluwer Academic Publishers.

[Arc01] ARCHER, TOM: *Inside C#*. Microsoft Press, München, 2001. ISBN 0735612889.

[Bal96] BALZERT, HELMUT: *Lehrbuch der Software-Technik: Software-Entwicklung*. Spektrum Akademischer Verlag, 1996. ISBN 3-8274-0042-2.

[Bal98] BALZERT, HELMUT: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, 1998. ISBN 3-8274-0065-1.

[Bal99] BALZERT, HEIDE: *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Spektrum Akademischer Verlag, 1999. ISBN 3-8274-0285-9.

[BC90] BRACHA, GILAD und WILLIAM COOK: *Mixin-based Inheritance*. In: MEYROWITZ, N. (Herausgeber): *Proceedings of the Joint Conference on Object-Oriented Systems, Languages, and Applications and the European Conference on Object-Oriented Programming (OOPSLA-ECOOP '90)*, Band 25, Nr. 10, Seiten 303–311. ACM SIGPLAN Notices, Oktober 1990.

[BD77] BACHMANN, C. W. und M. DAYA: *The Role Concepts in Data Models*. In: *Proceedings of the 3rd International Conference on Very Large Data Bases (VLDB)*, Seiten 464–476, 1977.

[Bec96] BECK, KENT: *Smalltalk Best Practice Patterns*. Prentice-Hall, 1996. ISBN 013476904X.

[Bec00] BECK, KENT: *Extreme Progrmming Explained: Embrace Change*. Addison-Wesley, 2000. ISBN 0-201-61641-6.

[Ben56] BENINGTON, H. D.: *Production of Large Computer Programs*. In: *Proceedings of the ONR Symposium on Advanced Programming Methods for Digital Computers*, Seiten 15–27, 1956.

[BF99] BAUMGART, JÖRG und BERND FREISLEBEN: *Ein Modell zur Erzeugung minimaler applikationsspezifischer Betriebssystemstrukturen durch dynamische Adaption*. In: CAP, C., W. ERHARD und W. KOCH (Herausgeber): *Architektur von Rechensystemen: Systemarchitektur auf dem Weg ins 3. Jahrtausend (ARCS '99 gemeinsam mit der APS '99)*, Seiten 155–166. VDE Verlag, 1999.

[BG95] BERTINO, ELISA und GIOVANNA GUERRINI: *Objects with Multiple Most Specific Classes*. In: *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP '95)*, LNCS 952, Seiten 102–126. Springer-Verlag, 1995.

[BGK+97] BÄUMER, DIRK, GUIDO GRYCZAN, ROLF KNOLL, CAROLA LILIENTHAL, DIRK RIEHLE und HEINZ ZÜLLIGHOVEN: *Framework Developement for Large Systems*. Communications of the ACM (CACM), 40(10):52–59, 1997.

[BK99] BÆKDAL, LARS KIRKEGAARD und BENT BRUUN KRISTENSEN: *Aggregation from Multiple Perspectives by Roles*. In: *Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS PACIFIC 99)*, Melbourne, Australia, 1999.

[Boe81] BOEHM, BARRY W.: *Software Engineering Economics*. Prentice-Hall, 1981. ISBN 0138221227.

[Boo87] BOOCH, GRADY: *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin-Cummings, 1987.

[Boo91] BOOCH, GRADY: *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Redwood City, 1991. ISBN 0-8053-5340-2.

[Boo94] BOOCH, GRADY: *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Redwood City, 2. Auflage, 1994.

[BR98] BÄUMER, DIRK und DIRK RIEHLE: *Product Trader*. In: MARTIN, ROBERT C., DIRK RIEHLE und FRANK BUSCHMANN (Herausgeber): *Pattern Languages of Program Design 3*, Seiten 29–46. Addison Wesley, 1998. ISBN 0-201-31011-2.

[BRSW97] BÄUMER, DIRK, DIRK RIEHLE, WOLF SIBERSKI und MARTINA WULF: *The Role Object Pattern*. In: *Proceedings of the 1997 Conference on Pattern Languages of Programming (PLoP '97)*, Monticello, Illinois, USA, 1997.

[BRSW99] BÄUMER, DIRK, DIRK RIEHLE, WOLF SIBERSKI und MARTINA WULF: *Role Object*. In: HARRISON, NEIL, BRIAN FOOTE und HANS ROHNERT (Herausgeber): *Pattern Languages of Program Design 4*, Seiten 15–31. Addison-Wesley, 1999. ISBN 0-201-43304-4.

[BS81] BRONSTEIN, I. und K. SEMENDJAJEW: *Taschenbuch der Mathematik*. Harry Deutsch, Thun, Frankfurt/Main, 20. Auflage, 1981.

[BW77] BOBROW, D. und T. WINOGRAD: *An Overview of KLR, A Knowledge Representation Language*. Cognitive Science, 1(1), 1977.

[Cab01] CABRI, GIACOMO: *Role-based Infrastructures for Agents*. In: *The 8th Workshop on Future Trends of Distributed Computing Systems (FTDCS 2001)*, Bologna, Italien, 2001.
http://www.agentgroup.unimo.it/MOON/papers/FTDCS01.pdf.

[CE00] CZARNECKI, KRZYSZTOF und ULRICH W. EISENECKER: *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000. ISBN 0-201-30977-7.

[CKC99] CONSTANZA, PASCAL, GÜNTHER KNIESEL und ARNIM B. CREMERS: *Lava – Spracherweiterungen für Delegation in Java*. In: CAP, C. H. (Herausgeber): *Java-Informations-Tage 1999 (JIT '99)*, Informatik Aktuell, Seiten 233–242, Düsseldorf, Deutschland, 1999. Springer-Verlag. ISBN 3-540-66464-5
http://javalab.cs.uni-bonn.de/data2/papers/darwin/lava.jit99.pdf.

[CLZ02a] CABRI, GIACOMO, LETIZIA LEONARDI und FRANCO ZAMBONELLI: *Modeling Role-based Interactions for Agents*. In: *Workshop on Agent-oriented methodologies at OOPSLA '02*, Seattle, USA, November 2002. http://www.agentgroup.unimo.it/MOON/papers/OOPSLA02.pdf.

[CLZ02b] CABRI, GIACOMO, LETIZIA LEONARDI und FRANCO ZAMBONELLI: *Separation of Concerns in Agent Applications by Roles*. In: *Proceedings of the International Workshop on Aspect Oriented Programming for Distributed Computing (AOPDCS); The 22nd International Conference on Distributed Systems (ICDCS-2002)*, Seiten 430–438, Wien, Österreich, Juli 2002. http://aopdcs.enst-bretagne.fr/cabri.ps.

[Cus89] CUSUMANO, M.A.: *The Software Factory: A Historical Interpretation*. IEEE Software, Mai 1989.

[DA98] DYSON, PAUL und BRUCE ANDERSON: *State Patterns*. In: MARTIN, ROBERT C., DIRK RIEHLE und FRANK BUSCHMANN (Herausgeber): *Pattern Languages of Program Design 3*, Seiten 125–142. Addison Wesley, 1998. ISBN 0-201-31011-2.

[Dem03] Homepage of the Demeter Project, Northeastern University, Boston, MA, 2003. http://www.ccs.neu.edu/research/demeter.

[DMSW96] DROSDOWSKI, GÜNTHER, WOLFGANG MÜLLER, WERNER SCHOLZE-STUBENRECHT und MATTHIAS WERMKE (Herausgeber): *Der Duden: Das Standardwerk zur deutschen Sprache*. Bibliographisches Institut & F. A. Brockhaus AG, Mannheim, 21. Auflage, 1996.

[ecl03] ECLIPSE.ORG, Mai 2003. http://www.eclipse.org.

[EF97] EVANS, ERIC und MARTIN FOWLER: *Specification Patterns*. In: *Proceedings of the 1997 Conference on Pattern Languages of Programming (PLoP '97)*, Monticello, Illinois, USA, 1997.

[EN94] ELMASRI, RAMEZ und SHAMKANT B. NAVATHE: *Fundamentals of Database Systems*. The Benjamin/Cumming Publication Company, Inc., 2. Auflage, 1994. ISBN 0-8053-1753-8.

[Eng93] ENGESSER, HERMANN (Herausgeber): *DUDEN Informatik: Ein Sachlexikon für Studium und Praxis*. Dudenverlag, 1993. ISBN 3-411-05232-5.

[Ern02] ERNST, ERIK: *Call by Declaration*. In: BLACK, ANDREW P., ERIK ERNST, PETER GROGONE und MARKKU SAKKINEN (Herausgeber): *Proceedings of the Inheritance Workshop at ECOOP '02*. Number 12 in Publications of Information Technology Institute. University of Jyväskylä, Juni 2002. http://www.cs.jyu.fi/~sakkinen/inhws/papers/Ernst.pdf.

[Fow97a] FOWLER, MARTIN: *Analysis Patterns: Reusable Object Modells*. Addison Wesley Longman, 1997. ISBN 0-201-89542-0.

[Fow97b] FOWLER, MARTIN: *Dealing with Roles*. In: *Proceedings of the 1997 Conference on Pattern Languages of Programming (PLoP '97)*, Monticello, Illinois, USA, 1997.

[Fri95] FRICK, ANDREAS: *Der Software-Entwicklungsprozeß – Ganzheitliche Sicht: Grundlagen zu Entwicklungs-Prozeß-Modellen*.
Carl Hanser Verlag, München, Wien, 1995. ISBN 3-446-17777-9.

[Gam98] GAMMA, ERICH: *Extension Object*. In: MARTIN, ROBERT C., DIRK RIEHLE und FRANK BUSCHMANN (Herausgeber): *Pattern Languages of Program Design 3*, Seiten 79–88. Addison Wesley, 1998. ISBN 0-201-31011-2.

[GB80] GOLDSTEIN, I. P. und D. G. BOBROW: *Descriptions for a Programming Environment*. In: BALZER, ROBERT (Herausgeber): *Proceedings of the 1st Annual National Conference on Artificial Intelligence*. AAAI Press/MIT Press, August 1980.

[Ghe02] GHELLI, GIORGIO: *Foundations for Extensible Objects with Roles*. Information and Computation, 175(1), Mai 2002.

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDIS: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995. ISBN 0-201-63361-2.

[GKS90] GOTTLOB, G., G. KAPPEL und M. SCHREFL: *Semantics of object-oriented data models – The evolving algebra approach*. In: SCHMIDT, J. W. und A. A. STOGNY (Herausgeber): *Next Generation System Technology*, LNCS 504, Seiten 144–160. Springer-Verlag, 1990.

[GR83] GOLDBERG, ADELE und DAVID ROBSON: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983. ISBN 0-201-11371-6.

[GR89] GOLDBERG, ADELE und DAVID ROBSON: *Smalltalk-80: The Language*. Addison-Wesley, 1989. ISBN 0-201-13688-0.

[Gri98] GRIFFEL, FRANK: *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt.verlag, 1998. ISBN 3-932588-02-9.

[GSR96] GOTTLOB, G., M. SCHREFL und B. RÖCK: *Extending Object-Oriented Systems with Roles*. ACM TOIS, 14(3):268–296, 1996.

[GWR00] GOLL, JOACHIM, CORNELIA WEISS und PETER ROTHLÄNDER: *Java als erste Programmiersprache: Java 2 Plattform*. B. G. Teubner Stuttgart, 2000.
ISBN 3-519-12642-7.

[HHG90] HELM, RICHARD, IAN M. HOLLAND und DIPAYAN GANGOPADHYAY: *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*. ACM SIGPLAN Notices, 25(10):303–311, 1990.

[HK99] HITZ, MARTIN und GERTI KAPPEL: *UML@Work*. dpunkt.verlag, Heidelberg, 1999.

[HO93] HARRISON, WILLIAM und HAROLD OSSHER: *Subject Oriented Programming (A Critique of Pure Objects)*. In: PAEPCKE, ANDREAS (Herausgeber): *Proceedings of the 1993 Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '93)*, Band 28, Nr. 10, Seiten 411–428, Washington, USA, Oktober 1993. ACM SIGPLAN Notices.

[Hür94] HÜRSCH, WALTER L.: *Should Superclasses be Abstract?* In: TOKORO, M. und R. PARESCHI (Herausgeber): *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP '94)*, LNCS 821, Seiten 12–31. Springer-Verlag, 1994.

[IBM] IBM RESEARCH: SUBJECT-ORIENTED PROGRAMMING GROUP: *Subject-oriented Programming and Design Patterns*.
http://www.research.ibm.com/sop/sopcpats.htm.

[JCJÖ92] JACOBSON, I., M. CHRISTERSON, P. JONSSON und G. ÖVERGAARD: *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison Wesley, Wokingham, 1992. ISBN 0-201-54435-0.

[JW98] JOHNSON, RALPH und BOBBY WOOLF: *Type Object*. In: MARTIN, ROBERT C., DIRK RIEHLE und FRANK BUSCHMANN (Herausgeber): *Pattern Languages of Program Design 3*, Seiten 47–65. Addison Wesley, 1998. ISBN 0-201-31011-2.

[Ken98] KENDALL, ELIZABETH A.: *Aspect-oriented Programming for Role Models*. In: *Proceedings of the Aspect-oriented Programming Workshop at ECOOP '98*, Brüssel, Belgien, 1998.
http://trese.cs.utwente.nl/aop-ecoop98.

[Ken99a] KENDALL, ELIZABETH A.: *Agent Roles and Aspects*. In: *Proceedings of the Aspect-oriented Programming Workshop at ECOOP '99*, Lissabon, Portugal, 1999.
http://trese.cs.utwente.nl/aop-ecoop99.

[Ken99b] KENDALL, ELIZABETH A.: *Role Model Designs and Implementations with Aspect-oriented Programming*. In: *Proceedings of the 1999 Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '99)*, Seiten 353–369, Denver, Colorado, USA, 1999. ACM Press.

[Ken00a] KENDALL, ELIZABETH A.: *Aspect-oriented Programming in AspectJ™*. Evolve 2000, Sidney, März 2000.
http://www.pscit.monash.edu.au/~kendall/selectedpapers.htm.

[Ken00b] KENDALL, ELIZABETH A.: *Role Modeling for Agent System Analysis, Design, and Implementation*. IEEE Concurrency, 8(2), 2000.

[Ken01] KENDALL, ELIZABETH A.: *Agent Software Engineering with Role Modelling*. In: CIANCARINI, PAOLO und MICHAEL J. WOOLDRIDGE (Herausgeber): *Agent-Oriented Software Engineering: First International Workshop (AOSE 2000)*, LNCS 1957, Seiten 163–169, Limerick, Ireland, 2001. Springer-Verlag.
ISBN 3-540-41594-7.

[KHH+01] KICZALES, GREGOR, ERIK HILSDALE, JIM HUGUNIN, MIK KERSTEN, JEFFREY PALM und WILLIAM G. GRISWOLD: *An Overview of AspectJ*. In: KNUDSEN, JØRGEN LINDSKOV (Herausgeber): *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01)*, LNCS 2072, Seiten 327–353. Springer-Verlag, 2001.

[KIL+97] KICZALES, GREGOR, JOHN IRWIN, JOHN LAMPING, JEAN-MARC LOINGTIER, CRISTINA VIDEIRA LOPES, CHRIS MAEDA und ANURAG MENDHEKAR: *Aspect-Oriented Programming*. In: AKŞIT, MEHMET und SATOSHI MATSUOKA (Herausgeber): *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, LNCS 1241, Seiten 220–242. Springer-Verlag, 1997.

[Kir01] KIRCHER, MICHAEL: *Akte XP: Hintergründe vom „eXtreme Programming"*. Java Spektrum, 1(29):15–17, 2001.

[KK95] KIM, W. und W. KELLY: *On View Support in Object-Oriented Database Systems*. In: KIM, W. (Herausgeber): *Modern Database Systems*, Seiten 108–129. Addison-Wesley, 1995.

[KM96] KRISTENSEN, BENT BRUUN und DANIEL MAY: *Activities: Abstractions for Collective Behavior*. In: COINTE, PIERRE (Herausgeber): *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '96)*, LNCS 1098, Seiten 472–501, Linz, Austria, 1996. Springer-Verlag.

[KMSW89] KÖSTER, K.-H., G. MÜLLER, S. SCHIEWE und M. WEBER: *CooL Reference Manual*. ITHACA Report, ITHACA.NIXDORF.89.L1.1, 1989.

[Kni96] KNIESEL, GÜNTHER: *Objects don't migrate! Perspectives on Objects with Roles*. Report IAI-TR-96-11, Institut für Informatik III, Universität Bonn, April 1996. http://javalab.cs.uni-bonn.de/data2/papers/darwin/roles.IAI-TR-96-11.pdf.

[Kni00] KNIESEL, GÜNTHER: *Dynamic Object-Based Inheritance with Subtyping*. Doktorarbeit, Institut für Informatik III, Universität Bonn, Juli 2000. http://javalab.cs.uni-bonn.de/data2/papers/darwin/darwinDiss.pdf.

[KØ94] KRISTENSEN, BENT BRUUN und KASPER ØSTERBYE: *Conceptual Modeling and Programming Languages*. ACM SIGPLAN Notices, 29(9):81–90, 1994.

[KØ96a] KRISTENSEN, B. B. und KASPER ØSTERBYE: *Roles: Conceptual Abstraction Theory and Practical Languages Issues*. Theory and Practice of Object Systems (TAPOS), 2(3):143–160, 1996.

[KO96b] KRISTENSEN, BENT BRUUN und JOHNNY OLSSON: *Roles & Patterns in Analysis, Design and Implementation*. In: *Proceedings of the 3rd International Conference on Object-oriented Information Systems (OOIS '96)*, London, England, 1996.

[KØ96c] KRISTENSEN, BENT BRUUN und KASPER ØSTERBYE: *A Conceptual Perspective on the Comparison of Object-Oriented Programming Languages*. ACM SIGPLAN Notices, 31(2):42–54, 1996.

[KP88] KRASNER, GLENN E. und STEPHEN T. POPE: *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. Journal of Object-Oriented Programming, 1(3):26–49, August 1988.

[Kri95] KRISTENSEN, BENT BRUUN: *Object-Oriented Modeling with Roles*. In: *Proceedings of the 2nd International Conference on Object-oriented Information Systems (OOIS '95)*, Dublin, Ireland, 1995.

[Kri97] KRISTENSEN, BENT BRUUN: *Subject Composition by Roles*. In: *Proceedings of the 4th International Conference on Object-oriented Information Systems (OOIS '97)*, Brisbane, Australia, 1997.

[KRS98] KAPPEL, GERTI, WERNER RETSCHITZEGGER und WIELAND SCHWINGER: *A Comparison of Role Mechanisms in Object-Oriented Modeling*. In: POHL, K., A. SCHÜRR und G. VOSSEN (Herausgeber): *Proceedings Modellierung '98, Bericht Nr. 6/98-I*, Seiten 105–109, Angewandte Mathematik und Informatik, Universität Münster, 1998.
http://www.computing.dcu.ie/˜lwang/papers/ComparsionRoleOO.pdf.

[Kru99] KRUCHTEN, PHILIPPE: *Der Rational Unified Process*. Addison-Wesley, 2. Auflage, 1999. ISBN 3827315433.

[KS91] KAPPEL, G. und M. SCHREFL: *Object/behaviour diagrams*. In: *Proceedings of the 7th International Conference on Data Engineering*, Seiten 530–539. IEEE Computer Society Press, 1991.

[KVN+89] KAPPEL, GERTI, JAN VITEK, OSCAR NIERSTRASZ, SIMON GIBBS, BETTY JUNOD, MARC STADELMANN und DENNIS TSICHRITZIS: *An Object-Based Visual Scripting Environment*. In: TSICHRITZIS, DENNIS (Herausgeber): *Object Oriented Developement*, Seiten 123–142. Centre Universitaire d'Informatique, Univ. of Geneva, 1989.

[LB02] LEE, JOON-SANG und DOO-HWAN BAE: *An enhanced role model for alleviating the role-binding anomaly*. Software Practice and Experience, 32:1317–1344, 2002.

[Lie96] LIEBERHERR, KARL J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996. ISBN 053494602-X.

[LL98] LEWIS, JOHN und WILLIAM LOFTUS: *Java Software Solutions: Foundations and Program Design*. Addison Wesley, 1998. ISBN 0-201-57164-1.

[LW93] LISKOV, BARBARA und JEANNETTE M. WING: *A New Definition of the Subtype Relation*. In: NIERSTRASZ, OSCAR M. (Herausgeber): *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, LNCS 707, Seiten 118–141. Springer-Verlag, 1993.

[LW94] LISKOV, BARBARA und JEANNETTE M. WING: *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6):1811–1841, 1994.

[Mey90] MEYER, BERTRAND: *Objektorientierte Softwareentwicklung*. Hanser Verlag, München, 1990.

[Mez96] MEZINI, MIRA: *Incremental Redefinition of Open Implementations*. In: ZIMMERMANN, CHRIS (Herausgeber): *Advances in Object-Oriented Metalevel Architectures and Reflection*, Kapitel 16, Seiten 265–290. CRC Press, 1996.
ISBN 0-8493-2663-X.

[Mez98] MEZINI, MIRA: *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publishers, 1998. ISBN 0-7923-8313-3.

[MM01] MAURO, JIM und RICHARD MCDOUGALL: *Solaris Internals: Core Kernel Architecture*. Prentice Hall, 2001. ISBN 0-13-022496-0.

[MMN93] MADSEN, L., B. MØLLER-PEDERSEN und K. NYGAARD: *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993. ISBN 0-201-62430-3.

[MP88] MCMENAMIN, STEPHEN M. und JOHN F. PALMER: *Strukturierte Systemanalyse*. Hanser Verlag, 1988. ISBN 3446151664.

[ND95] NIERSTRASZ, O. und L. DAMI: *Component-Oriented Software Technology*. In: NIERSTRASZ, O. und D. TSICHRITZIS (Herausgeber): *Object-Oriented Software Composition*. Prentice Hall, 1995.

[NS98] NEHMER, JÜRGEN und PETER STURM: *Systemsoftware: Grundlagen moderner Betriebssysteme*. dpunkt-Verlag, 1998. ISBN 3-920993-74-8.

[Obj00] OBJECT MANAGEMENT GROUP: *Agent Technology: Green Paper*, 2000. http://www.objs.com/agent/agents_Green_Paper_v100.doc.

[Obj03] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language Specification: Version 1.5*, März 2003. http://www.omg.org/technology/documents/formal/uml.htm.

[Ode00] ODELL, JAMES: *Introduction to Agents*, 2000. http://www.objs.com/agent/agents_omg.pdf.

[OHE97] ORFALI, R., D. HARKEY und J. EDWARDS: *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1997.

[OL01] ORLEANS, DOUG und KARL LIEBERHERR: *DJ: Dynamic Adaptive Programming in Java*. In: YONEZAWA, AKINORI und SATOSHI MATSUOKA (Herausgeber): *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, REFLECTION 2001*, LNCS 2192, Seiten 73–80, Kyoto, Japan, 2001. Springer-Verlag. ISBN 3-540-42618-3.

[OT01a] OSSHER, HAROLD und PERI TARR: *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. In: *Proceedings of the Symposium on Software Architecture and Component Technology: The State of the Art in Software Developement*. Kluwer, 2001.

[OT01b] OSSHER, HAROLD und PERI TARR: *Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software*. Communications of the ACM (CACM), 44(10):43–50, Oktober 2001.

[Pap91] PAPAZOGLOU, MIKE P.: *Roles: A Methodology for Representing Multifaceted Objects*. In: *Proceedings of the International Conference on Database and Expert Systems Applications*, Seiten 7–12. Springer-Verlag, 1991.

[Par72] PARNAS, DAVID LORGE: *On the Criteria to be Used in Decomposing Systems into Moduls*. Communications of the ACM (CACM), 15(2):1053–1058, 1972.

[Par75] PARNAS, DAVID LORGE: *Software Engineering or Methods for Multi-Person Construction of Multi-Version Programs*. In: HACKL, CLEMENS (Herausgeber): *IBM Symposium: Programming Methodology*, LNCS 23, Seiten 225–235. Springer-Verlag, 1975.

[Per90] PERNICI, BARBARA: *Objects with Roles*. In: *Proceedings of the ACM-IEEE Conference on Office Information Systems (COIS)*, Seiten 205–215, 1990.

[PJ00] PAPAZOGLOU, MIKE P. und KRÄMER BERND J.: *Modeling Object Dynamics*. In: PAPAZOGLOU, MIKE P., STEFANO SPACCAPIETRA und ZAHIR TARI (Herausgeber): *Advances in Object Oriented Data Modeling*, Seiten 195–217. MIT Press, 2000.

[PJA94] PAPAZOGLOU, MIKE P., KRÄMER B. J. und BOUGUETTAYA A.: *On the Representation of Objects with Polymorphic Shape and Behaviour*. In: LOUCOPOULOS, P. (Herausgeber): *13th International Conference on The Entity Relationship Approach*, LNCS 881, Seiten 223–240. Springer-Verlag, 1994.

[PK97] PAPAZOGLOU, MIKE P. und B. J. KRÄMER: *A Database Model for Object Dynamics*. The VLDB Journal, 6(2):73–96, Mai 1997.

[PS96] PFISTER, CUNO und CLEMENS SZYPERSKI: *Why Objects Are Not Enough*. In: *Proceedings, First International Component Users Conference (CUC '96)*, München, Deutschland, 1996. SIGS.

[PTMA89] PRÖFROCK, ANNA-KRISTIN, DENNIS TSICHRITZIS, GERHARD MÜLLER und MARTIN ADER: *ITHACA: An Integrated Toolkit for Highly Advanced Computer Applications*. In: TSICHRITZIS, DENNIS (Herausgeber): *Object Oriented Developement*, Seiten 321–344. Centre Universitaire d'Informatique, Univ. of Geneva, 1989.

[RAB+92] REENSKAUG, TRYGVE, EGIL P. ANDERSEN, ARNE JØRGEN BERRE, ANNE HURLEN, ANTON LANDMARK, ODD ARILD LEHNE, ELSE NORDHAGEN, EIRIK NÆSS-ULSETH, GRO OFTEDAL, ANNE LISE SKAAR und PÅL STENSLET: *OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems*. Journal of Object-Oriented Programming (JOOP), Seiten 27–41, Oktober 1992.

[RBP+91] RUMBAUGH, JAMES R., MICHAEL R. BLAHA, WILLIAM PREMERLANI, FREDERICK EDDY und WILLIAM LORENSEN: *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991. ISBN 0-13-629841-9.

[Ree97] REENSKAUG, TRYGVE: *Role Modeling Enters the Main Stream*. Object EXPERT, Januar 1997.

[RG98] RIEHLE, DIRK und THOMAS GROSS: *Role Model Based Framework Design and Integration*. In: *Proceedings of the 1998 Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '98)*, Seiten 353–369, Vancouver, Canada, 1998. ACM Press.

[Rie96] RIEHLE, DIRK: *Describing and Composing Patterns Using Role Diagrams*. In: MÄTZEL, KAI-UWE und HANS-PETER FREI (Herausgeber): *Proceedings of the 1996 Ubilab Conference*, Zürich, Schweiz, 1996. Universitätsverlag Konstanz. http://www.riehle.org/computer-science/research/1996/ubilab-woon-1996.pdf.

[Rie97] RIEHLE, DIRK: *Composite Design Patterns*. In: *Proceedings of the 1997 Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '97)*, Seiten 218–228, Atlanta, Georgia, USA, 1997. ACM Press.

[Rie98] RIEHLE, DIRK: *Bureaucracy*. In: MARTIN, ROBERT C., DIRK RIEHLE und FRANK BUSCHMANN (Herausgeber): *Pattern Languages of Program Design 3*, Seiten 163–185. Addison Wesley, 1998. ISBN 0-201-31011-2.

[Rie00] RIEHLE, DIRK: *Framework Design: A Role Modeling Approach*. Doktorarbeit, ETH Zürich, 2000.
http://www.riehle.org/computer-science/research/dissertation/diss-a4.pdf.

[Roy70] ROYCE, W. W.: *Managing the Development of Large Software Systems*. In: *Proc. IEEE WESTCON (Nachdruck in: Proceedings of the 9th International Conference on Software Engineering, Seiten 328-338, Monterey, USA, 1987)*, Seiten 1–9, Los Angeles, USA, 1970.

[RS91] RICHARDSON, JOEL und PETER SCHWARZ: *Aspects: Extending Objects to Support Multiple, Independent Roles*. In: CLIFFORD, J. und R. KING (Herausgeber): *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Band 20 der Reihe *SIGMOD Record*, Seiten 298–307, Denver, Colorado, USA, 1991. ACM Press.

[RS03] RAJAN, HRIDESH und KEVIN SULLIVAN: *Need for Instance Level Aspect Language with Rich Pointcut Language*. In: *Proceedings of the Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT 2003) at AOSD 2003*, Boston, MA, USA, 2003.

[RWL95] REENSKAUG, TRYGVE, PER WOLD und ODD ARILD LEHNE: *Working wirh Objects: The OOram Software Engineering Method*. Prentice Hall, 1995.
ISBN 1-884777-10-4.

[RWL01] REENSKAUG, TRYGVE, PER WOLD und ODD ARILD LEHNE: *Working wirh Objects: The OOram Software Engineering Method*, Februar 2001.
http://folk.uio.no/trygver/documents/book11d.pdf.

[Sam97] SAMETINGER, JOHANNES: *Software Engineering with Reusable Components*. Springer-Verlag, 1997. ISBN 3540626956.

[SB86] STEFIK, M. und D. G. BOBROW: *Object-Oriented Programming: Themes and Variations*. AI Magazine, Januar 1986.

[Sch96] SCHOENFELD, ARI: *Domain Specific Patterns: Conversions, Persons and Roles, and Documents and Roles*. In: *Proceedings of the 1996 Conference on Pattern Languages of Programming (PLoP '96)*, Allerton Park, Illinois, USA, 1996.

[Sci89] SCIORE, EDWARD: *Object Specialization*. ACM Transactions on Information Systems, 7(2):103–122, April 1989.

[SDNB03] SCHÄRLI, NATHANAEL, STÉPHANE DUCASSE, OSCAR NIERSTRASZ und ANDREW P. BLACK: *Traits: Composable Units of Behaviour*. In: CARDELLI, LUCA (Herausgeber): *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '03)*, LNCS 2743, Seiten 248–274, Darmstadt, Germany, 2003. Springer-Verlag.

[SGM02] SZYPERSKI, CLEMENS, DOMINIK GRUNTZ und STEPHAN MURER: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2. Auflage, 2002. ISBN 0-201-74572-0.

[Sie98] SIELSKE, ANDRÉ: *Vergleichende Implementierung eines Rollenkonzepts in verschiedenen Zielsprachen*. Diplomarbeit, Institut für Informatik III, Universität Bonn, November 1998.
http://javalab.cs.uni-bonn.de/data2/papers/darwin/daAndre.ps.zip.

[SN88] SCHREFL, M. und E. J. NEUHOLD: *Object class definition by generalization using upward inheritance*. In: *Proceedings of the IEEE 4th International Conference on Data Engineering*, Seiten 4–13. IEEE, 1988.

[Som01] SOMMERVILLE, IAN: *Software Engineering*. Addison-Wesley, 6. Auflage, 2001. ISBN 0-201-39815-X.

[SP94] SCHRÖDER-PREIKSCHAT, WOLFGANG: *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994.

[SR02] SEKHARAIAH, CHANDRA K. und JANAKI D. RAM: *Object Schizophrenia Problem in Modeling Is-role-of Inheritance*. In: BLACK, ANDREW P., ERIK ERNST, PETER GROGONE und MARKKU SAKKINEN (Herausgeber): *Proceedings of the Inheritance Workshop at ECOOP '02*. Number 12 in Publications of Information Technology Institute. University of Jyväskylä, Juni 2002.
http://www.cs.jyu.fi/~sakkinen/inhws/papers/Sekharaiah.pdf.

[ST01] SCHREFL, MICHAEL und THOMAS THALHAMMER: *Using Roles in Java*, 2001.
http://www.dke.uni-linz.ac.at/papers/Schr01b.pdf.

[ST03a] SCHREFL, MICHAEL und THOMAS THALHAMMER: *Using Roles in Java*. Software-Practice and Experience (SP&E), 2003. Zur Veröffentlichung angenommen.

[ST03b] SCHREFL, MICHAEL und THOMAS THALHAMMER: *Das* JAVA *Role API*, 8. Mai 2003.
http://www.dke.uni-linz.ac.at/roles/index.html.

[Ste00] STEINMANN, FRIEDRICH: *On the Representation of Roles in Object-Oriented and Conceptual Modelling*. Data & Knowledge Engineering, 35(1):83–106, 2000.

[Str00] STROUSTRUP, BJARNE: *Die* C++*-Programmiersprache. Deutsche Übersetzung der Special Edition*. Addison-Wesley, Mai 2000. ISBN 382731660X.

[Tai96] TAIVALSAARI, ANTERO: *On the Notion of Inheritance*. ACM Computing Surveys, 28(3):438–479, September 1996.

[Tam99] TAMAI, TETSUO: *Objects and Roles: Modelling based on the Dualistic View*. Information and Software Technology, 41(14):1005–1010, 1999. http://www.graco.c.u-tokyo.ac.jp/~tamai/pub/roles.pdf.

[Tam02] TAMAI, TETSUO: *Evolvable Programming based on Collaboration-Field and Role Model*. In: *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02)*, Seiten 1–5, Orlando, Florida, USA, 2002. ACM. http://www.graco.c.u-tokyo.ac.jp/~tamai/pub/iwpse02-kn.pdf.

[TGML98] TU, M. T., F. GRIFFEL, M. MERZ und W. LAMERSDORF: *A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents*. In: ROTHERMEL, KURT und FRITZ HOHL (Herausgeber): *Mobile Agents: Second International Workshop, MA '98*, LNCS 1477, Seiten 222–236, Stuttgart, Deutschland, 1998. Springer-Verlag. ISBN 3-540-64959-X.

[The03a] THE ASPECTJ TEAM: *The AspectJ ™ Programming Guide*, Juni 2003. http://www.eclipse.org/aspectj.

[The03b] THE DARWIN PROJECT, Juli 2003. http://javalab.cs.uni-bonn.de/research/darwin/index.html.

[The03c] THE DARWIN PROJECT: *Lava II Distribution*, Juli 2003. http://javalab.cs.uni-bonn.de/research/darwin/download.html.

[TM97] THAYER, R. und DORFMANN M. (Herausgeber): *Software Requirements Engineering*. IEEE Computer Society Press, 2. Auflage, 1997. ISBN 0818677384.

[TO00] TARR, PERI und HAROLD OSSHER: *Hyper/J ™ User and Installation Manual*, 2000. http://www.research.ibm.com/hyperspace.

[TOHS99] TARR, PERI, HAROLD OSSHER, WILLIAM HARRISON und STANLEY M. SUTTON: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In: *Proceedings International Conference on Software Engineering (ICSE) '99*, Seiten 107–119. ACM Press, 1999.

[US87] UNGAR, R. und R.B. SMITH: *Self: The Power of Simplicity*. In: *Proc. OOPSLA '87*, Seiten 227–240. ACM Press, 1987.

[UT01] UBAYASHI, NAOYASU und TETSUO TAMAI: *Separation of Concerns in Mobile Agent Applications*. In: YONEZAWA, AKINORI und SATOSHI MATSUOKA (Herausgeber): *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, REFLECTION 2001*, LNCS 2192, Seiten 89–109, Kyoto, Japan, 2001. Springer-Verlag. ISBN 3-540-42618-3.

[Ven97] VENNER, BILL: *The architecture of aglets*. JavaWorld, April 1997. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood_p.html.

[VHN96a] VAN HILST, MICHAEL und DAVID NOTKIN: *Using C++ Templates to Implement Role-Based Designs*. In: FUTATSUGI, KOKICHI und SATOSHI MATSUOKA (Herausgeber): *Object Technologies for Advanced Software: Second JSSST International Symposium, ISOTAS '96, Proceedings*, LNCS 1049, Seiten 22–37, Kanazawa, Japan, März 1996. Springer-Verlag. ISBN 3-540-60954-7.

[VHN96b] VAN HILST, MICHAEL und DAVID NOTKIN: *Using Role Components to Implement Collaboration-Based Designs*. In: *Proceedings of the 1996 Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '96)*, Seiten 359–369, San Jose, California, USA, 1996. ACM Press.

[Wan89] WAND, YAIR: *A proposal for a formal model of object*. In: KIM, WON und FREDERICK H. LOCHOVSKY (Herausgeber): *Object-Oriented Concepts, Databases and Applications*, ACM Press Frontier Series, Kapitel 21, Seiten 537–559. ACM Press, 1989. ISBN 0201144107.

[WBJ90] WIRFS-BROCK, REBECCA und RALPH E. JOHNSON: *A Survey of Current Research in Object-Oriented Design*. Communications of the ACM, 33(9):105–124, September 1990.

[WCL96] WONG, RAYMOND K., H. LEWIS CHAU und FREDERIK H. LOCHOVSKY: *DOOR: A dynamically object-oriented data model with roles*. Technical Report HKUST-CS96-12, Hong Kong University of Science & Technology, 1996.

[WCL97] WONG, RAYMOND K., H. LEWIS CHAU und FREDERIK H. LOCHOVSKY: *A Data Model and Semantics of Objects with Dynamic Roles*. In: *Proceedings of the 13th International Conference on Data Engineering (ICDE '97)*, Seiten 402–411. IEEE, 1997.

[WWW99] WWW CONSORTIUM: *XSL Transformations (XSLT) Version 1.0*, November 1999. http://www.w3org/TR/xslt.

[WWW00] WWW CONSORTIUM: *Extensible Markup Language (XML) 1.0 (Second Edition)*, Oktober 2000. http://www.w3org/TR/2000/REC-xml-20001006.pdf.

[WWW03] WWW CONSORTIUM: *The Extensible Stylesheet Language Family (XSL)*, 2003. http://www.w3org/Style/XSL.

[ZBGK01] ZUSER, WOLFGANG, STEFAN BIFFL, THOMAS GRECHENIG und MONIKA KÖHLE: *Software-Engineering mit UML und dem Unified Process*. Pearson Studium, 2001. ISBN 3-8273-7027-2.

[ZF98] ZHAO, LIPING und TED FOSTER: *A Pattern Language of Transport Systems (Point and Route)*. In: MARTIN, ROBERT C., DIRK RIEHLE und FRANK BUSCHMANN (Herausgeber): *Pattern Languages of Program Design 3*, Seiten 409–430. Addison Wesley, 1998. ISBN 0-201-31011-2.

[ZK00] ZHAO, LIPING und ELIZABETH KENDALL: *Role Modelling for Component Design*. In: *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS 2000)*, 2000.
http:www.computer.org/proceedings/hicss/0493/04938/04938048.pdf.

[ZK01] ZHAO, LIPING und ELIZABETH KENDALL: *Role Modelling for Component Design*. In: *Proceedings TOOLS33 Technology of Object Oriented Languages and Systems*, Seiten 312–323. IEEE Computer Society Press, 2001.
ISBN 0-7695-0731-X.

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit zur Erlangung des akademischen Grades **Dr.-Ing.** mit dem Titel **Analyse, Entwurf und Generierung von Rollen- und Variantenmodellen** selbständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel erstellt zu haben. Weiterhin erkläre ich, vor dieser Arbeit keinen Promotionsversuch unternommen zu haben.

Darmstadt, den 3. September 2003

Jörg Baumgart

Lebenslauf

Geburtsdatum	13. Juni 1961	
Geburtsort	Darmstadt	
Familienstand	ledig	
Eltern	Prof. Helmut Baumgart, Hochschullehrer (FH), Architekt Brigitte von Streit, verw. Baumgart, geb. Rothhardt	
Schulausbildung	09.1967 – 07.1971	Goethe-Schule, Darmstadt
	09.1971 – 12.1979	Georg-Büchner-Schule, Darmstadt Vorgezogene Abiturprüfung
Hochschulausbildung	04.1980 – 07.1980	Chemie-Studium Technische Hochschule Darmstadt
	10.1980 – 06.1986	Informatik-Studium Technische Hochschule Darmstadt Abschluß: Diplom-Informatiker
Berufstätigkeit	12.7.82 – 27.8.82 21.2.83 – 31.3.83 20.2.84 – 30.3.84	Werkstudent Boehringer Mannheim Allgemeine Biometrie
	09.1986 – 11.1988	Software-Entwickler Telenorma Darmstadt Bereich Öffentliche Technik
	12.1988 – 05.1994	Wissenschaftlicher Mitarbeiter Technische Hochschule Darmstadt Fachbereich Informatik Fachgebiet Betriebssysteme
	06.1994 – 12.1994	Freiberufliche Tätigkeit Werkvertrag mit der TH Darmstadt Lehraufträge an der BA Mannheim Datenbankseminare
	01.1995 – 09.1999	Wissenschaftlicher Mitarbeiter Universität Gesamthochschule Siegen FB Elektrotechnik und Informatik Fachgruppe Parallele Systeme
	seit 10.1999	Dozent an der Berufsakademie Mannheim (University of Cooperative Education)